# SYSTEM, METHOD, USES, PRODUCTS, PROGRAM PRODUCTS, AND BUSINESS METHODS FOR DISTRIBUTED INTERNET AND DISTRIBUTED NETWORK SERVICES OVER MULTI-TIERED NETWORKS



## 1. EXTENDED ENVIRONMENT DATA STRUCTURE FOR DISTRIBUTED DIGITAL ASSETS OVER A MULTI-TIERS COMPUTER NETWORK

## 2. SERVER SYSTEM AND METHOD FOR DISCOVERING DIGITAL ASSETS IN ENTERPRISE INFORMATION SYSTEMS

## 3. SERVER SYSTEM AND METHOD FOR EXPORTING DIGITAL ASSETS IN ENTERPRISE INFORMATION SYSTEMS

## 4. SERVER SYSTEM AND METHOD FOR DISTRIBUTING AND SCHEDULING MODULES TO BE EXECUTED ON DIFFERENT TIERS OF A NETWORK

## 5. DATA STRUCTURE, ARCHITECTURE APPARATUS, AND PROGRAM PRODUCT CAPABLE OF BEING DISTRIBUTED TO AND EXECUTED ON DIFFERENT NETWORK DEVICES AND ON VARIOUS COMPUTER PLATFORMS AND ENVIRONMENTS

## 6. SYSTEM AND METHOD FOR TRANSACTIONAL DEPLOYMENT J2EE WEB COMPONENTS, ENTERPRISE JAVA BEAN COMPONENTS, AND APPLICATION DATA OVER MULTI-TIERED COMPUTER NETWORKS

## 7. SYSTEM AND METHOD FOR DISTRIBUTING ASSETS TO MULTI-TIERED NETWORK NODES

## 8. METHOD AND SYSTEM FOR DEPLOYING AN ASSET OVER A MULTI-TIERED NETWORK

**9. SYSTEM AND METHOD FOR TRANSLATING AN ASSET FOR DISTRIBUTION OVER MULTI-TIERED NETWORKS**

**10. SYSTEM AND METHOD FOR SYNCHRONIZING ASSETS ON MULTI-TIERED NETWORKS**

**11. METHOD AND SYSTEM FOR DEPLOYING AN ASSET OVER A MULTI-TIERED NETWORK, PARTICULARLY TO A TARGETED NODE**

**12. SYSTEM AND METHOD FOR ADJUSTING THE DISTRIBUTION OF AN ASSET OVER A MULTI-TIERED NETWORK**

**13. SYSTEM AND METHOD FOR BRIDGING ASSETS TO NETWORK NODES ON MULTI-TIERED NETWORKS**

**14. METHOD AND SYSTEM FOR DEPLOYING A CHANGED ASSET OVER A MULTI-TIERED NETWORK**

**15. SYSTEM, METHOD, AND DATA STRUCTURE FOR PACKAGING ASSETS FOR PROCESSING AND DISTRIBUTION ON MULTI-TIERED NETWORKS**

APPENDICES:

**A. SYSTEM AND METHOD FOR COLLABORATION USING WEB BROWSERS**

**B. SYSTEM AND METHOD FOR TRANSACTIONAL AND FAULT-TOLERANT DISTRIBUTION OF DIGITAL ASSETS OVER MULTI-TIERED COMPUTER NETWORKS**

PRIORITY DOCUMENTS

This application claims priority to the provisional U.S. patent application number 60/229,685, entitled "Distributed Internet Services Provisional Filing", filed on 9/1/00 to Chen et al. This provisional application is herein incorporated by reference in its entirety.

RELATED REFERENCES

The following references are related to this patent application and are herein incorporated by reference in their entirety:

provisional U.S. patent application number 60/236,864 entitled "Distributed Internet Server" to Pace et al., filed 9/29/00;

provisional U.S. patent application number 60/237,179 entitled "Business Plan and Business Plan Executive Summary" to Chen et al., filed 10/2/00;

provisional U.S. patent application number 60/254,377 entitled "Distributed Internet Services provisional filing II" to Pace et al., filed 12/8/00;

provisional U.S. patent application number 60/262,288 entitled "Data Structure, Architecture, Apparatus, and Program Product Capable of Being Distributed to and Executed on Different Network Tiers and on Various Computer Platforms and Environment" to Pace et al., filed 1/17/01;

U.S. Patent application number XXXX, entitled "Extended Environment Data Structure for Distributed Digital Assets Over Multi-Tiered Networks", to Pace et al., filed XXX;

U.S. Patent application number XXXX, entitled "Server System and Method for Discovering Digital Assets in Enterprise Information Systems", to Bobick et al., filed xxx;

U.S. Patent application number XXXX, entitled "Server System and Method for Exporting Digital Assets in Enterprise Information Systems", to Pace et al. filed xxxx;

U.S. Patent application number XXXX, entitled "System and Method for Transactional Deployment J2EE Web Components, Enterprise Java Bean Components, and Application Data over Multi-tiered Computer Networks", to Pace et al., filed on xxx;

U.S. Patent application number xxxx, entitled "Server System and Method for Distributing and Scheduling Modules to be Executed on Different Tiers of a Network", to Pace et al. filed on xxx;

U.S. Patent application number xxxx, entitled "Data Structure, Architecture, Apparatus, and Program Product Capable of Being Distributed to and Executed on Different Network Devices and on Various Computer Platforms and Environments", to Pace et al. filed on xxx;

U.S. Patent application number xxxx, entitled "System and Method for Distributing Assets to Multi-Tiered Network Nodes , to Pizzorni et al. filed on xxx;

U.S. Patent application number xxxx, entitled "Method and System for Deploying An Asset Over a Multi-Tiered Network", to Pace et al. filed on xxx;

U.S. Patent application number xxxx, entitled "System and Method for Translating an Asset for Distribution Over Multi-Tiered Networks (Processing)" to Pace et al. filed on xxx;

U.S. Patent application number xxxx, entitled "System and Method for Synchronizing Assets on Multi-Tiered Networks, to Pace et al. filed on xxx;

U.S. Patent application number xxxx, entitled "Method and System for Deploying an Asset Over a Multi-Tiered Network"' to Pace et al. filed on xxx;

U.S. Patent application number xxxx, entitled "System and Method for Adjusting the Distribution of an Asset Over a Multi-Tiered Network", to Pace et al. filed on xxx;

U.S. Patent application number xxxx, entitled "System and Method for Bridging Assets to Network Nodes on Multi-Tiered Networks", to Pace et al. filed on xxx;

U.S. Patent application number xxxx, entitled **streaming**, to Pace et al. filed on xxx;

U.S. Patent application number xxxx, entitled "System, Method, and Data Structure for Packaging Assets for Processing and Distribution on Multi-Tiered Networks", to Bobick et al. filed on xxx;

U.S. Patent application number xxxx, entitled System and Method for Transactional and Fault-Tolerant Distribution of Digital Assets Over Multi-Tiered Computer Networks, to Bobick et al. filed on xxx;

U.S. Patent application number XXXX, entitled "System and Method for Collaboration Using Web Browsers", to Chen et al. filed on xxx; and

PCT Patent application number xxxx, entitled "System, Method, Uses, Products, Program Products, and Business Methods for Distributed Internet and Distributed Network Services", to Pace et al. filed on xxx.

Priority is also claimed to these documents for any matter not disclosed in the filing serial number 60/229,685, incorporated above.

## FIELD OF THE INVENTION

1. This invention relates to the field of distributing network (e.g. web and server applications) systems, sub systems, applications, sub applications, components, modules, functions, over one or more computer networks. More specifically, the invention relates to an extended environment data structure comprising part of a digital asset.

2. This invention relates to the field of distributing network systems, sub systems, applications, sub applications, components, modules, functions, over one or more computer

networks. More specifically, the invention relates to discovering unique digital assets in enterprise information system for export over a multi-tier network system.

3. This invention relates to the field of distributing network (e.g. web and server applications) systems, sub systems, applications, sub applications, components, modules, functions, over one or more computer networks. More specifically, the invention relates to exporting digital assets in enterprise information system for distribution over a multi-tier network system.

4. This invention relates to the field of distributing network (e.g. web) applications over one or more computer networks. More specifically, the invention relates to organizing, scheduling and distributing web application components/modules and package data structures to be executed on clients within a multi-tier network system.

5. This invention relates to the field of distributing network (e.g. web) applications over one or more computer networks. More specifically, the invention relates to a class of data structures capable of being organized, scheduled, and distributed for execution on different nodes of a network.

6. This invention relates to the field of distributing network (e.g. web and server applications) systems, sub systems, applications, sub applications, components, modules, functions, over one or more computer networks. More specifically, the invention relates to the transactional deployment of J2EE web components, enterprise java bean components, and application data over multi-tiered networks.

7. This invention relates to the field of distributing network (e.g. web and server applications) systems, sub systems, applications, sub applications, components, modules, functions, over one or more computer networks. More specifically, the invention relates to a system and method for transactional and fault-tolerant distribution of digital assets over multi-tiered computer networks.

## BACKGROUND OF THE INVENTION

Network applications have evolved over time to take on a multi-tiered client and server arrangement (architecture). Typically, one or more server computers are connected through their network interfaces by one or more networks to one or more client computers. Networks include data networks (such as the Internet), voice networks (such as telephone), wired or wireless networks, and any combination of these used to communicate data, voice, programs, general content, and/or other information. Networks can be local to a company or organization, like a local area network (LAN) and intranet, or they can expand over large geographies, like wide area networks (WAN), that even interconnect other networks. Of course, one widely used and developing network is the Internet, which includes the World Wide Web (WWW). The WWW uses web browser software running on the client computers of the network to execute certain web-based applications. These web-based applications include pages that are served from one or more of the web servers on the WWW in HTML

format.

Many applications on the Internet, and other network environments, use a module or modules of software called "middleware". Broadly, middleware can be any computer software function that is performed between one or more clients and one or more host systems such as database and web servers. However, typically, middleware runs on servers that operate between the clients and other servers in a network. Examples of these other servers include Oracle Database, IBM DB2 and IBM CICS. Middleware is often used to execute certain computer programs which are meant to off load processing from these other servers, to preprocess information for client computers, and/or to perform a set of functions or services that are commonly needed for certain kinds of applications. Some examples of functions or services that are typically performed by "middleware" would be transaction monitoring and coordination, server load-balancing, host fail-over and other application level services.

A typical enterprise information system (EIS) is comprised of client computers, middleware servers and database servers. Web servers are included within the EIS when web browser based clients must be served via the Internet/Intranet. EIS's are generally known and include application programs that perform the functions required by any given business and/or organization. Non limiting examples of an EIS include: on-line customer order entry; on-line retail/wholesale sales, marketing, and inventory systems; enterprise supply chain management systems; product and/or content distribution systems (e.g. television, home video); on-line financial systems (e.g., mortgage applications, investing, stock trading, loan application, credit card accounts); service providing systems (including medical, legal, real estate, engineering, education, distance leaning, technical support); on-line human resource and pay roll services; the on-line banking systems (e.g., deployed by a bank or other financial institutions and/or the retail banking systems used internally by bank personnel); airline reservation systems; and any other general way of transacting business over a network.

Often these functions/application programs are made of parts. For example, an application program can be made of components, modules, or functions (see Figure 1F), which in turn are made of objects. Typically, the component, module or function also includes either an expressed or implied order in which to execute the respective objects in the component, module, or function. This order can be shown or defined by an execution graph. Again, the execution graph can be implied by the "calling structure" of the program.

Execution of one or more of these components, modules, functions and/or entire applications can be performed at various locations over these networks. This well known type of program execution is called distributed programming. The advantage of distributed programming is mostly to utilize the distributed computing resources over the network to achieve the best performance. Performance can be gauged using certain criteria such as execution time, and controlled using certain middleware applications such as fault-tolerance and load balancing. Important computing resources such as CPUs, network bandwidth, software functions and data storage must be well managed in order to achieve customary and generally known system requirements usually referred to as reliability, availability and scalability (RAS).

Distributed computing can allow programs to run faster because the work of the program is divided among multiple computer systems. Also, specific tasks in the program can be executed on a computer that has the facilities to best execute these tasks. For example, a mathematically intensive task could be run on a computer that has a particularly fast process for executing mathematical problems and application programs that support a large variety of mathematic functions.

However, distributed programming often fails if the communication among the processors involved gets too complicated. Programs have to be developed and installed to insure that data among the processors is coherent. Some systems cannot tolerate noise or spurious signals on the network. Delays in receiving data from one or more processors could slow the entire system. In order to be distributed, application programs must be written so that tasks can be broken apart and the results of these tasks accurately combined. This often greatly adds to project development costs, assuming that these objectives can be accomplished at all. Communication between the various computers on the network and the programs/protocols the computers are using must be compatible.

Often the network is thought of as being divided into tiers where each of these components, modules, or functions is executed. These tiers are commonly divided by functional (or logical) and/or physical computing levels or sub-tiers. The advantage of dividing the network application system into multiple tiers is to facilitate the development and deployment of the various computing resources. Some times tiers are thought of as physical locations where components, modules, or functions of programs are executed. For example, some components, modules or functions can be executed on the EIS tier or middleware tier, while other components, modules, or functions are executed on the client computers (the client tier). Alternatively, tiers of the network can be divided logically, such as on a small human resource system, where the client and server part of the components, modules, or functions are all on one computer, but logically the components, modules, or functions are still developed and deployed based on the client and the server tier being separate tiers.

Network tiers can also be combinations of physical and logical tiers. For example, take an on-line banking system that is comprised of a client computer, middleware servers, and various backend database systems. Suppose the client, middleware and database systems are physically separate computer systems (tiers). The middleware tier can be subdivided into logical tiers such as a web server, an application server, and a transaction server.

In much of the existing middleware, objects used are highly interdependent and defined by the function(s) of the middleware. Some well known middleware objects include: Sun Microsystem's Java Server Page (JSP) and Enterprise Java Bean (EJB). The JSP object executes programs, based on requests from one or more clients. The EJB object executes certain programs that are pre-packaged into an "Enterprise Java Bean" format. Other objects include: general data files, general programs, and general multimedia content files (e.g., text, video, sound, and voice content). (Java Server Page and Enterprise Java Bean are trademarks of the Sun Microsystem Corporation.)

It is often necessary for various servers and clients to communicate even though they may have different run-time environments (i.e., are running different application programs like middleware) and are running on different platforms (i.e., have different hardware and operating systems). Generally, servers and clients communicate using well-known protocols like HTTP over TCP/IP. Other network communication protocols include Internet Interoperable Protocol (IIOP) that permits communication between different computer platforms over the network. One example of a technology that uses IIOP would be the Common Object Request Broker Architecture (CORBA). (CORBA is a trademark of The Open Group.) At a high level CORBA specifies many standards involving application level communication among disparate applications and computing platforms.

The prior art discloses some "open" architectures that permit programmers to develop code that will have general use in a networking environment. Some of these architectures permit communication between programs executing on different systems, different platforms or environments, and even using different programming languages over the network (and network tiers.) The open architecture encourages development of applications that can be used generally with a flexibility to interact with any other architecture based program (component, module, function or object) without regard to what, where, or on what system the other application parts exist or execute.

One such open architecture system is called JINI (a Trademark of Sun Microsystems). JINI uses JAVA (also a Trademark of Sun Microsystems) technology to wrap these otherwise incompatible programs, particularly driver programs of input/output devices so that these devices can be plugged into a JINI compatible network and operate and communicate with any other device on the network. For example, JINI can be used to permit any pervasive device on a wireless network to communicate with any other JINI compatible pervasive device that comes within the communication range of the wireless network.

Refer to Figure 1, partially including Figures 1A-1E, each a block diagram of a prior art middleware computer platform.

Figure 1A shows a general middleware computer system 160 with well known computer hardware 100, a general network operating system (like Windows NT – a trademark of the Microsoft corporation) 102, a middleware platform (Microsoft Commerce Server) 104, a transactional operating system (like Microsoft Transaction Server - MTS) 106, and any given application program (like on-line ticketing sales) 108.

Figure 1B is a block diagram of a generally known Java middleware platform 170 that has computer hardware 100 and a network operating system 102, as before. A middleware platform which supports Enterprise Java Beans (EJB) 114 runs on the network operating system 102 that allows Java application programs 118 to run on the system 170.

Figure 1C is a block diagram of a generally known CORBA middleware platform 180 that has computer hardware 100 and a network operating system 102, as before. The CORBA middleware platform 124 permits general application programs 120 to operate on this

platform. Non-limiting examples of these application programs 120 include Java, C++, COBOL, and Smalltalk.

Figure 1D is a block diagram of a generally known Windows middleware system 190 that operates on windows compatible hardware 100. A windows DNS (COM/MTS) or MTS system is a middleware system 134 available from the Microsoft Corporation that permits any general application program 120 to run on the platform 190.

Figure 1E is a block diagram of a generally known system 195 that uses hardware 100, a network operating system 102, and a middleware program 144 called TUXEDO. TUXEDO is a trademark of the BEA Systems, Inc. This platform 195 runs application programs 146 written in the C programming language.

Figure 1F is a representation showing a prior art hierarchical relationship among system and application parts. The largest part is a system 105F that must contain one or more complete applications 108. The system 105F also can contain one or more sub system 106F that each in turn comprise one or more applications 108. The application 108 is a group of one or more computer programs. Sub applications 110F are parts of applications 108. Some applications 108 and/or sub applications 110F comprise one or more components 120F.

A component 120F exists at a significant layer within an application 108. Typically, a component 120F is part of a distributed system that interacts with its environment by exchanging message/information with other components 120F and/or applications (108, 110F). Generally, components 120F comprise runnable (or executable) and non-runnable parts. The runnable/executable parts of components 120F are generally called modules 130. Modules 130 in turn comprise one or more functions 140F also known as routines 140F or methods 140F.

Middleware, and for that matter, other prior art programs that function in a network environment, often need to communicate information between logical and/or physical functions in the network. For example, data or programs (e.g. objects) might need to be passed to a program or component, module, or function executing on the same machine as the application program. On the other hand, this information might have to be passed across the network to components,, modules, functions, sub applications, or applications that are running on completely different computers. The prior art has various ways of addressing this problem. Some prior art passes information "by value" between components, modules, functions or applications. Thus, information needed is passed, e.g. in a "call statement" or header file to the component, module, function or application requiring the information. Other information, e.g. results from a calculation can be passed back in the same manner. Other prior art, uses replication. In replication, programs and data are copied from one machine (computer) to a second machine where they are executed in an "island" mode.

Some prior art, e.g. Castanet (a trademark of Marimba, Inc.), is able to package and deploy business applications to computers over the network. Other prior art include content distribution systems, such as those marketed by Akamai, Inc., physically locate caching

servers throughout the world to cache content from web sites and provide more local delivery to end user (clients). Similar systems include Digital Island, Ltd's global distribution network, called Footprint, that pushes certain application content through the network to servers closer to the end-user. Inktomi Traffic Server (a trademark of the Inktomi Corporation) is a network cache platform that delivers certain application content to servers across the network.

Several terms and concepts are defined in the prior art of software analysis, design, and languages. Software systems can be composed of one or more applications. These applications can be assembled from one or more components, modules, functions or objects.

In software written using object-oriented techniques, many modules further have a one-to-one correspondence with a class in the particular object-oriented language. A class or set of classes may also be considered a component if the class or set of classes meets the requirements of a specified component system. Examples of component systems are: COM, CORBA, EJB, ActiveX, XPCOM, and Java Beans.

Classes may be composed of one or more functions or procedures optionally coupled along with one or more variables. The functions or procedures are referred to as methods, while the variables are referred to as data members.

At runtime, classes are instantiated into objects, which are distinct entities, separate from the definition of the object, that is, the class. The data members represent the data or state of the object. The methods characterize the behavior of the object.

Build systems transform collections of non-runnable computer files into runnable computer modules and assembles them into components and applications. Build system cannot identify or export required digital assets 240 on existing Enterprise Information System (EIS). Build systems also cannot identify runtime execution and data dependencies in previously installed EIS applications. Build system generally contain incremental linkers which establishes runtime relationships among modules of object code and are an improvement over regular linkers because they re-link only changed modules on command.

Archive utilities (like Zip, Gzip, and Tar files) are used for distributing and storing files. These files contain one or more program and data files. Usually the files "archived" in a zip-like archive are compressed to save space. Archive files make it easy to group files and make transporting and copying these files faster. Typically, archive utilities examine the file types of the files to be zipped and invoke a file type specific compression subroutine to compress the file and add it to the archive.

Other types of software examine computer files and invoke rules based on file type to achieve specific results. Specifically, virus scan software will examine executable programs and based on one or more rules, determine whether there is a virus in the executable routine of the programs. Virus scan software (such as McAfee) can not be used and is not meant to be used to discover particular software, package the software, and then distribute the software over a

network.

Software which may be classified as "enhanced" build systems (for example, Starbase) control versioning of code and static elements of software products during development, as well as deployment of the completed products of development using various transport mechanisms to destination user computing platforms. Such enhanced build systems are designed to control and deploy only work in progress, ancillary products, and the completed products of development, and the inventory of code, static, and ancillary elements managed by such systems is rigorously constrained and must be rigorously constrained to those purposes. Such enhanced build systems cannot be used and are not meant to be used to discover particular software, package the software, and then distribute the software over the internet.

The prior art also discloses specifications for deployment of web components (particularly J2EE web components), enterprise java bean (EJB) components, and J2EE application data. The J2EE specification leaves methods of transactional deployment of J2EE of web and EJB components to application server products that otherwise comply with the J2EE specification. There is no provision in the J2EE specification for transactional deployment of J2EE application data.

## PROBLEMS WITH THE PRIOR ART

Using different computing environments and platforms creates many communication and operability problems on the network. For example, many computing environments (including middleware environments) only can operate with programs with which they are specifically designed to operate. Much of the prior art is unable to communicate or share information or programs with any general platform or computing environments. Much of the prior art cannot distribute programs and/or data over one or more networks so that the programs can be executed and the data used on any given platform or environment. Where this distribution occurs, it is only possible with the expenditure of considerable resources and highly trained system designers.

The prior art needs to be able to distribute data, programs, and portions of programs in a more efficient way over various tiers of a network to operate on any general platform or environment.

Another variation of the problem involves the explanation of middleware's intra-tier distribution versus inter-tier distribution. Middleware application servers are targeted at defining tiers of functionality. These tiers may scale within the tier, but not necessarily utilizing the processing power available in other tiers.

Much of the prior art middleware is constrained. Typically, middleware is only used with a particular EIS and is designed specifically for that EIS's platform and environment. Often this middleware operates in local area networks with 10/100 megabits of bandwidth or less. Most of this middleware cannot effectively function in a wide area network environment, or on the

Internet, where bandwidth capabilities are often more restrictive. This middleware cannot communicate with computer systems that do not use the same communication protocols for which the middleware was designed.

Much of the middleware typically operates between the EIS web server and the EIS database management system (DBMS). The result is that the performance of the middleware is limited by the performance of the EIS web server and/or the EIS DBMS.

Much of the middleware does not work with components, modules or functions that are designed to execute on a platform/environment different than that of the EIS for which the middleware was designed. Therefore, this middleware can't organize, schedule, and/or distribute applications outside of the EIS. This prior art middleware cannot enable the execution of any general component, module, function, and/or application program to be executed on any general computer with any general platform/environment, nor does this middleware suggest how this might be done. The prior art middleware cannot distribute application programs and/or components, modules or functions for execution over different tiers of a network, nor has the prior art recognized the need to do this.

Some prior art architectures, like JINI, permit communication between computers with different platforms/environments. However, much of this communication is used to enable standard interface functions like print, read data, etc. These architectures are not capable of decomposing complex application programs, of the sort found on EIS's, and recomposing these application programs so that they can be executed on any given platform. These prior art architectures cannot organize, schedule, and/or distribute application programs and/or components, modules, or functions across many tiers of a network so that these application programs/components, modules or functions can be executed on any general platform/environment.

Much of the prior art cannot automatically identify and extract sub applications, components, modules, functions, and specific files and data structures from legacy programs located on an EIS in order to export these application parts to other machines connected to the EIS through one or more networks. In addition, the prior art generally fails to identify these application parts by type so that the application parts can be processed in such a way to include the necessary information and structures particular to the type so that the application part can be transformed and/or executed on various tiers of the network.

## OBJECTS OF THE INVENTION

1. An object of this invention is an improved data structure for defining digital assets.

1. An object of this invention is an improved data structure for defining digital assets for packaging, distribution, deployment, execution, synchronization, and/or lifecycle management over multi-tiered networks.

2. An object of this invention is an improved system and method for discovering and/ or identifying and extracting digital assets from an EIS or other sources so that EIS/source systems, sub systems, applications, sub applications, components, modules, or functions, and/or objects can be packaged, distributed, deployed, executed, synchronized, and/or managed through a lifecycle in a distributed manner.

2.An object of this invention is an improved system and method for discovering and/or identifying, extracting, packaging, distributing, deploying, and/or exporting, web applications, components, modules or functions and/or objects over the Internet.

2.An object of this invention is an improved system and method for discovering and/or identifying types of digital assets that are extracted from one or more EISs and identifying those digital assets according to their respective type so that the digital asset can be exported, e.g., in packages, from the EIS as a means of distribution, deployment, and/or execution over tiers of the network.

3. An object of this invention is an improved system and method for extracting and exporting digital assets of an EIS so that EIS systems, sub systems, applications, sub applications, components, modules, or functions, and/or objects can be packaged, distributed, deployed, executed, synchronized, and/or managed through a lifecycle over a multi-tiered network.

3.An object of this invention is an improved system and method for extracting and exporting as a means of distributing, deploying, and/or executing web applications, components, modules or functions and/or objects over the Internet.

3.An object of this invention is an improved system and method for extracting and exporting types of digital assets that are extracted from one or more EIS/source and identifying those digital assets according to their respective type so that the digital asset can be exported, e.g., in packages, from the EIS/source to be distributed, deployed, executed, synchronized, and/or managed through a lifecycle over tiers of the network.

4. An object of this invention is an improved system and method for distributing assets throughout a network so that applications, components, and/or objects can be executed in a distributed manner.

4. An object of this invention is an improved system and method for distributing execution of web applications, components, and/or objects over the Internet.

4. An object of this invention is an improved system and method for enabling faster, more secure, and reliable execution of applications, components, and/or objects over a network.

4. An object of this invention is an improved system and method for distributing a new package architecture to be executed on different tiers of a network and on various computer platforms.

4. An object of this invention is an improved server and server method capable of distributing and scheduling general modules to be executed on different tiers of the network.

4. An object of this invention is an improved system and method for distributing and scheduling modules to be executed on different tiers of the network to enable various business transactions and/or general collaboration over the network.

5. An object of this invention is an improved data structure, system, and program product for distributing assets throughout a network so that applications, data, components, and/or objects can be executed and accessed in a distributed manner.

5. An object of this invention is an improved data structure, system, and program product for enabling faster, more secure, and reliable execution of and access to applications, data, components, and/or objects over a network.

5. An object of this invention is an improved component architecture that is capable of being distributing and executed on different devices of a network and on various computer platforms.

6. An object of this invention is an improved system and method for transactional deployment of assets over multi-tiered networks.

6. An object of this invention is an improved system and method for transactional deployment of assets over multi-tiered networks where the assets are either J2EE web components or J2EE EJB components along with J2EE application data.

## SUMMARY OF THE INVENTION

1. Extended Environment for digital asset

The present invention is a data structure, program product, and product of manufacture that has an extended environment (EE) data structure that is part of a digital asset. The digital asset is capable of being transmitted over one or more multi-tiered networks. The data structure has one or more common descriptors that provide a unique identification of the digit asset on the networks. There are also one or more asset dependency descriptors that identify one or more associated digital assets. Associated digital assets are associated with the digital asset by means of a joint membership as parts of a whole. The asset further has one or more target server dependencies descriptors that identify a base execution environment on one or more target computers. (The base execution environment is required to execute the digital asset on the respective target computer. The base execution environment comprises zero or more other digital assets deployed to the respective target computer from one or more of the Enterprise Information Systems (EIS) or other sources.) In an alternative embodiment, one or more EIS server dependencies descriptors are included that identify an EIS execution

environment on the respective EIS/source from which the asset resides. In alternative embodiments, other descriptors are included in the extended environment data structure.

2. Discover

The present invention is a system, method, article of manufacture, and a computer program product that identifies (discovers) member objects of one or more computer system parts in an Enterprise Information System (EIS) or other sources while establishing at least one relationship (e.g., topographical) among the member objects. The invention begins by traversing one or more computer file systems of the EIS/source to find one or more of the member objects. For each member object found, a digital asset identifier of the found member object is placed in an intermediate representation. The intermediate representation is a graph with nodes and edges. Each of the digital asset identifiers corresponds to one of the nodes of the graph. The edges represent the relationship. A digital asset is created from the member object by placing the member object in a logic/data section of the digital asset and attaching an extended environment data structure to the logic/data section. The digital asset is stored in an asset inventory container object. This may be repeated for each found member object until the intermediate representation fully describes the computer system part and the asset inventory container object is a complete inventory of the digital assets of interest in the computer system part. Additional structures describing attributes of the digital asset created can also be constructed. Further, the descriptive information related to the digital asset may be placed in the respective extended environment data structure.

3. Export

The present invention is a system, method, article of manufacture, and a computer program product that locates and categorizes identified member objects of one or more computer system parts in an Enterprise Information System (EIS) or other sources for export to either a packaging process or to another computer system over tiers of one or more networks. The invention begins by traversing an intermediate representation of one or more parts of a computer system while applying one or more context rules to determine a context of the parts. The context may be either a standard specified context or a non-specified context. If a standard specified context is determined, a directed search is performed to acquire any of the set of runnable and/or non-runnable member objects in one or more locations in an Enterprise Information System (EIS) or other sources as identified in the intermediate representation and specified by the context. If a non-specified context is determined, an implicit traversal search is performed for any of the set of runnable and/or non-runnable member objects in one or more locations in an Enterprise Information System (EIS)/source identified in the intermediate representation. One or more of the set of runnable and/or non-runnable member objects are then accessed at their respective locations in the EIS/source. A preliminary package specification is made for the accessed set of the runnable and/or non-runnable member objects. Digital assets in an asset inventory that correspond to the respective runnable and non-runnable member objects, are listed in the preliminary package specification and are

updated by adding one or more export descriptors to the extended environment of the respective digital assets. In an alternative embodiment, one or more entries in an asset definition data structure corresponding to each of the respective digital assets are updated.

4. CDS

The present invention is a computer system and method used as a server connected to one or more networks through one or more network interfaces. The server, called a component distribution server (CDS) system, comprises a package specification process that receives one or more packages from the network. These received packages are subparts of one or more application programs from one or more enterprise information systems (EIS). The packages have one or more assets. Each asset has an asset type and two or more asset layers. A first asset layer is a logic/data layer with information that embodies the function of the asset (asset function). The second asset layer is an extended environment layer that is a subset of the EIS and that has portions of the EIS necessary to support the respective logic/data layer. The CDS system also has a process adapter process that translates one or more of the assets layers so that the asset can perform the asset function on one or more target base environments of one or more target computers. Further, the CDS system has a target process that changes one or more of the layers of the asset in order to provide specific information for one or more specific target computers. An asset that is translated by the process adapter process and changed by the target process is called a transformed asset.

5. Package

The present invention is a novel package structure, system for using this package structure, and program product containing this package structure. The package structure can be stored in one or more memories (e.g., as an article of manufacture or on an EIS or client system) and/or distributed over one or more networks. The package structure has one or more first assets, each with a first logic/data part and a first extended environment part. These first assets can have a relational data asset type or a static content asset type. The package structure also has one or more second assets, each having a second logic/data part and a second extended environment part. The second assets have a presentation component asset type, typically used to generate content.

In alternate preferred embodiments, the package structure also has one or more third assets, each with a third logic/data part and a third extended environment part. The third assets have a transaction component asset type or a static content asset type.

In another alternative preferred embodiment, one or more forth assets exist. The four assets are selected so that the package has at least one asset from each of the following asset types: relational data, presentation component, transaction component, and static content.

In an alternative preferred embodiment, the first assets have a static content asset type and the second assets have a presentation component asset type. Further, the optional third asset can have a relational data asset type and/or a transaction component asset type.

6. J2EE Component Transactional Deployment

The present invention is a computer system, method, article of manufacture, and computer program product for transactional deployment of one or more components over a multi-tier network. The system has one or more J2EE application servers stored on one or more memories of the system and capable of being executed by one or more central processing unit (CPUs). One or more J2EE applications can be executed on the J2EE application servers. One or more J2EE application containers are contained within the J2EE application server. One or more J2EE application container components are contained within the respective J2EE application containers. One or more Java EJB containers are contained within the J2EE application container. One or more Java web containers are contained within the J2EE application container. One or more delivered J2EE components are delivered to the J2EE application server over one or more tiers of one or more the network. There are one or more logical connections to one or more DBMS located on the networks. In addition, there is a sphere of control encompassing the J2EE application server, the J2EE applications, the J2EE application containers, the EJB containers, the web containers, the J2EE application container components, the delivered J2EE components, and the logical connections. The sphere of control manages a transactional deployment of the delivered J2EE components and an update of the databases to keep the data consistent with the J2EE application.

7. Deploy - Manifest

An exemplary method and/or exemplary embodiment of the present invention distributes an asset to a multi-tiered network node. An asset may represent network and/or application components (e.g., data, objects, applications, program modules, etc.) that may be distributed among the various resources of the network. In an embodiment, a pending notice is received from a distribution server. If the notice indicates that at least one asset is pending (i.e., awaiting deployment), an asset descriptor manifest is received from the distribution server. The asset descriptor manifest identifies at least one asset to be deployed to the node, and includes an offset associated with the asset identifier. The asset descriptor manifest is stored in a memory on the node. A fragment, associated with the asset, is received and stored in the memory. The offset associated with the asset is marked with the end of the fragment, and another fragment, beginning at the offset, is then received. Additional fragments are received, and the offset updated, until the entire asset is deployed to the node. In an alternative embodiment, the entire asset is received in the first fragment. In another embodiment, multiple assets are received in the first fragment.

8. Deploy – Adaptor

The present invention provides a method and system for deploying assets to multi-tiered network nodes. An asset may represent network and/or application components (e.g., data, objects, applications, program modules, etc.) that may be distributed among the various

resources of the network. In one embodiment, a target node's environment may be adjusted before an asset is deployed to that target node. In an alternative embodiment, a target deployment adapter, associated with the asset, may be selected and deployed with the asset in order to allow the asset to operate in the target node environment. An implementation class, associated with the asset, may be inserted into the target node environment. An altered target deployment descriptor may also be inserted into the target node environment.

9. Process/Translation

The present invention provides a system and method for translating an asset for distribution to a multi-tiered network node. An asset may represent network and/or application components (e.g., data, objects, applications, program modules, etc.) that may be distributed among the various resources of the network. In an embodiment, an asset has a logic/data section and an extended environment section. The logic/data section defines a function of the digital asset along with the asset's type, while the extended environment section supports the function of the logic/data section within at least one source environment. The asset type is determined and a process asset adapter, associated with the asset type and a target environment, is selected. The asset is then translated into a processed asset having a processed extended environment section supporting the function of the logic/data section in the target environment.

10. Synchronization

An exemplary method and/or exemplary embodiment of the present invention synchronizes an asset over a multi-tiered network. An asset may represent network and/or application components (e.g., data, objects, applications, program modules, etc.) that may be distributed among the various resources of the network. Synchronization addresses the restoration of asset coherency in a distributed system, i.e. bringing changes made to assets on one distributed node into harmonization with changes made to assets on another distributed node. In an embodiment, a synchronization call having a data argument and an asset type is received, an adapter associated with the asset type is selected, and the data argument is passed to the

adapter. The asset type is determined, as well as a table associated with the asset type. A synchronization information object is retrieved from a target environment on a target node, and a synchronization asset is created based on the synchronization information. A connection is established between the target node and the asset's original source node, and the synchronization asset is sent from the target node to the source node.

11. Target

One embodiment of the present invention is a method of operating a computer system for targeting one or more digital assets on a distribution server connected to one or more networks so that the digital assets are compatible with one or more target nodes connected to the networks. The method includes examining the one or more digital assets to determine an asset type of the digital asset and, if the asset type is Relational Data (RD), retrieving one or more where clauses of the digital asset. The method further includes executing a token replacement operation on the where clause to create a transformed where clause and running a query on one or more tables specified in the digital asset using the transformed where clause, the query returning one or more returned records and the returned records correlating with the target node. The method further includes storing the returned record in the digital asset.

12. Adjust

The present invention provides a system and method for adjusting the distribution of an asset over a multi-tiered network. An asset may represent network and/or application components (e.g., data, objects, applications, program modules, etc.) that may be distributed among the various resources of the network. In an embodiment, a performance metric is received and input to a performance model. The model determines a network optimization and at least one change requirement based on the metric. A package specification is changed to reflect the requirement, which may specify that at least one asset should be packaged in at least one package. The change requirement is implemented and the package is distributed over the network.

13. Bridging

An exemplary method and/or exemplary embodiment of the present invention provides for bridging assets over a multi-tiered network. An asset may represent network and/or application components (e.g., data, objects, applications, program modules, etc.) that may be distributed among the various resources of the network. Generally, communications can be maintained between executable assets residing on different network nodes by bridging the execution context of the two nodes. In an embodiment, a mapping layer can be generated for assets that have run-time dependencies; the mapping layer uses a distribution system to bridge the execution context of a first environment with that of a second environment. The asset executing in the first environment is able to access another resource located in the second environment, even though the asset does not have local access to the resource in the second environment. A fault is detected when at least one asset deployed on a local node attempts to access at least one resource on a remote node through an application programming interface. The fault is then handled appropriately.

14. Streaming

One embodiment of the present invention is a method for distributing changes to digital assets across a network. The method includes determining an asset type of a first digital asset and comparing the first digital asset to a prior digital asset to determine one or more deltas, the prior digital asset being a prior version of the first digital asset and the delta being a difference between the first digital asset and the prior digital asset. The method further includes evaluating the one or more of the deltas with one or more criteria to determine if the one or more delta assets should be created, the delta asset being a second digital asset containing the respective delta, the criteria determined by the asset type. The method further includes that if the delta meets the criteria, creating the delta asset, and marking the delta asset as a first delta asset of the first digital asset.

15. EEP

The present invention provides a system, method, and data structure for packaging assets for processing and distribution over a multi-tiered network. An asset may represent

network and/or application components (e.g., data, objects, applications, program modules, etc.) that may be distributed among the various resources of the network. In an embodiment, the package structure includes at least one representation of an asset having a logic/data portion and an asset extended environment portion, and a package extended environment that includes package information associated with at least one asset.

## BRIEF DESCRIPTION OF THE FIGURES

Figure 1, comprising Figures 1A-1E, are block diagrams of various prior art middleware systems.

Figure 1F is a representation showing a prior art hierarchical relationship among system and application parts.

Figure 1G is a drawing showing the conceptual association of one or more system parts to one or more node/client/targets using an engagement table.

Figure 1H is a block diagram of one preferred engagement table.

Figure 2 is a block diagram of an asset.

Figure 2A is a diagram of an asset life cycle.

Figure 2B is a block diagram of a preferred extended environment (EE) of a digital asset.

Figure 2C is an UML diagram of one preferred embodiment of an asset class object.

Figure 3 is a block diagram of one preferred package content data structure that identifies which assets are packaged in which packages.

Figure 4 is a block diagram of an asset definition data structure.

Figure 5 is a block diagram of a deployable asset data structure.

Figure 6 is a block diagram of a client asset table.

Figure 7 is a block diagram of a client deployment queue.

Figure 8 is a block diagram of one preferred physically architecture of a network system that

uses the present invention.

Figure 9 is a block diagram of a preferred logical architecture of a network system using the present invention.

Figure 10 is a diagram of an example data structure showing a plurality of packages (package) and assets.

Figure 11 is a block diagram of an example data structure for package definition.

Figure 11A is a block diagram of a preferred package data structure with an extended environment for packages (EEP).

Figure 11B is a UML diagram showing the transitive part-whole associations between certain digital assets and certain packages.

Figure 12 is a flow chart of a package specification process.

Figure 12A is a block diagram of an example of a packaging queue.

Figure 13 is a flow chart of an asset packaging process.

Figure 14 is a flow chart of a client deployment process.

Figure 14A is a block diagram of an asset descriptor manifest data structure.

Figure 14B is a block diagram of an alternative client asset table, with versioning.

Figure 15 is a flow chart of a node registration process.

Figure 15A is a block diagram of a node registration specification.

Figure 16 is a flow chart of an EIS export adapter process.

Figure 16A is a flow chart of a version asset adapter process.

Figure 16B is a flow chart of an alternative preferred EIS export adaptor process.

Figure 17 is a flow chart of a client deploy adapter process.

Figure 17A is a block diagram of one preferred J2EE transactional deployment sphere of control.

Figure 17B is a flow chart of the implementation of one preferred J2EE transaction deployment within the sphere of control of Figure 17A.

Figure 18 is a flow chart of a process adapter process.

Figure 18A is a flow chart of a transform adapter process, a preferred embodiment of the process adapter process.

Figure 19 is a flow chart of a target adapter process.

Figure 20 is a flow chart of a synchronize adapter process.

Figure 21 is a flow chart of a discovery asset adapter process.

Figure 21A is a flow chart of an adjustment asset adapter process.

Figure 21B is a flow chart of an alternative preferred discovery asset adapter process.

Figure 21C is a diagram of a prior art graph structure, specifically a containment graph structure, used to establish membership relationships of digital assets.

Figure 21D is a block diagram of a preferred asset inventory.

Figure 22 is a flow chart of a publishing agent process.

Figure 23 is a flow chart of a subscriber agent process.

Figure 24 is a flow chart of a computational agent process.

Figure 25 is a flow chart of a caching agent process.

Figure 26 is a flow chart of an asset distribution adapter process.

Figure 27 is a flow chart of an asset streaming adapter process.

Figure 28 is a flow chart of a bridging process for bridging computational environments.

Figure 29 is a flow chart of a Quality of Service process.

Figure 30A is a flow chart of some example steps used by target/client nodes in the present asset distribution architecture.

Figure 30B is a flow chart of some preferred example steps performed by a server in the present asset distribution architecture.

Figure 30C is a flow chart of some preferred example steps used in package/application definition and distribution.

Figure 30D is a flow chart of some preferred example steps in asset distribution to computational environments and caches.

Figure 30E is a flow chart of some preferred example steps showing asset execution in a distributed manner across network tiers.

Figure 30F is a flow chart of some preferred example steps showing distribution of current assets.

Figure 31 is a block diagram showing some preferred example connections and asset distributions in various uses, e.g., pervasive uses, of the invention across tiers of the network.

## DETAILED DESCRIPTION OF THE INVENTION

One of the novel features of this disclosure is that a system part 100F, e.g. an enterprise application, is distributed and/or transformed over two or more tiers of a given network. The system part 100F can then be deployed, and/or cached, and/or executed on one or more remote target computer computing devices (target environment and/or platforms).

Note that in this disclosure, the terms node, client, and target all refer to computing devices attached to one or more networks that communicate in well-known ways with one or more Enterprise Information Systems (EIS). While these terms are used interchangeably without loss of generality, the term node more specifically refers to a network-centric view of such a computing device, the term client more specifically refers to an application-centric view of such a computing device, and the term target specifically refers to an invention-centric view of such a computing device.

Typically, these system parts 100F can be distributed easier and faster over the network than the entire system 105F from which the system part 100F originated. Distributing these relatively smaller system parts 100F uses fewer resources of the network and uses those resources more efficiently.

In some instances, the entire system 105F (sub systems 106F, and/or application 108) is too large and/or proprietary to be distributed to and/or executed on the target computing device. In a preferred embodiment of the present system, only the essential and/or non-proprietary system part(s) 100F need be distributed over the network and executed on one or more of the target computing devices (940 below). This allows large and/or proprietary programs from one or more source computers (910, 915 below) to execute in a client specific way close to and/or on the target computing device.

In a preferred embodiment, these relevant system parts 100F are distributed so that they are physically closer to the target(s). Since only the system parts 100F need to be distributed

close to the target(s), less storage is need at, or near, the target as compared to the case where the entire system 105F (sub system 106F and/or application 108F) is distributed. Distributing only the essential portions of the system 105F required by the target in closer physical proximity to the target results in a faster and more reliable execution at the target. This is because network errors and delays are reduced because much less of the network is used during the execution of the system parts 100F on the target.

A benefit of distributing these system parts 100F is that the integrity (i.e., the security, transactions, and coherence) of the source environment (e.g., EIS) can be maintained at one or more of the target/clients while the original system part 100F remains on the source (EIS). Therefore, the infrastructure to maintain the system part 100F can be reduced by locating the distributed system parts 100F at one or more of the clients since more of the client resources are used to support the source resources allowing the resources of the source to be used more efficiently.

In some preferred embodiments of this invention, system parts 100F are transformed to operate on remote target environments/platforms 940 that are different than the platforms and environments (source) 915 of their respective applications, e.g. an Enterprise Information System (EIS). For example, using the invention, a Solaris/DB2/WebLogic "source" application can be automatically sub partitioned into a sub application that is then executed on an NT/Hypersonic SQL/JBOSS/Tomcat "target" so that the sub application operates on a completely different platform and/or environment than the source application. Another example is using an HPUX/Oracle/Netscape Enterprise Server "source" and a MacIntosh/Jigsaw/Foxbase client which can be the target of a distribution .

Figure 1G is a drawing showing the conceptual association of one or more system parts to one or more node/client/targets using an engagement table. The association of system part 100F to target computing device 835 (see Figure 8) is required for the distribution of one or more system parts 100F to one or more target computing devices 835. This association, also described as an "engagement", can take many embodiments. In its most simple embodiment, one system part is engaged to one target computing device. In another embodiment, one system part is engaged to more than one target computing device. In another embodiment, many system parts are engaged to one target computing device . In another embodiment, many system parts are engaged to many target computing devices.

In alternative embodiments, one or more target computing devices 835 can be identified, addressed as, and/or organized into classes. Such classes can reflect, geographic, logical, business category-based or any other general class relationships. As is well known in the art, these classes can contain sub classes in a hierarchical relation. The target computing devices in these classes can be engaged together to one or more system parts 100F.

In alternative embodiments, one or more system parts 100F can be identified, addressed as, and/or organized into classes. Such classes can reflect technical requirement, business purpose, or any other general class relationships. As is well known in the art, these classes

can contain sub classes in a hierarchical relation. The system parts in these classes can be engaged together to one or more target computing devices 835.

In a preferred embodiment, these engagements are achieved through use of an engagement table data structure.

Figure 1H is a block diagram of one preferred engagement table 100H. The engagement table 100H contains a plurality of system part 100F to target computer device 835 pairs called engagement pairs 110H. Each engagement pair 110H represents a relationship between one system part 100F and one target computing device 835. One to many relationships are defined by multiple engagement pairs 110H.

Each engagement pair contains a part identifier 120H of the system part 100F. Example part identifiers 120H include system name, sub system name, application name, component name, module name, function name, a fully qualified address, and any other identifier capable of uniquely identifying the respective system part 100F on the network.

Each engagement pair also contains a target identifier 130H of the target computing device 835. Example target identifiers 130H include a target/client ID 1382, a client ID (1392, 1452), an internet protocol address (IP address), a network name address, node ID 1524, and/or any other identifier capable of uniquely identifying the respective target computer device 835 on the network.

A sub application 110F is that part of an application 108 that can be partitioned at a sub application boundary. Examples of sub application boundaries defining a runnable system part (100F) demarcated by: an application programming interface (API), component boundary, an Internet protocol boundary, and/or any other logical program division in the application, e.g. an edge of the application program call graph. Alternatively, such a sub application boundary can be defined to include: 1. an open API module, 2. a proprietary standard API component that is made generally available to a user/developer, or 3. an (some) extension framework. Also see the paragraph below.

Alternatively, the sub application boundary, or package boundary, can be defined by the boundaries of one or more of the assets (see below) in the package structure. These assets can have an asset boundary, defining the package boundary that includes any one or more of the following: an application programming interface (API) module, a component boundary, a Internet protocol function boundary, a logical program division in the application, one or more edges of an application program call graph for the application, an open standard API module, a proprietary standard API component, and an extension framework.

Non limiting examples of an open standard API used as part or all of a package (asset) boundary include any one or more of the following: one or more TCP/IP Standard Libraries, one or more Standard C/C++ Libraries, one or more CORBA Libraries, one or more Java servlets, one or more Java server pages, one or more enterprise Java beans (EJB), a JDBC (Java database connectivity), a JMS (Java Messaging Service), a HTML (Hypertext Markup

Language), an HTTP (Hypertext Transfer Protocol), and a WML (Wireless Markup Language).

Non limiting examples of a proprietary standard API used as part or all of a package (asset) boundary include any one or more of the following: a J2EE, one or more Microsoft Foundation Classes (MFC), one or more COM/DCOM (component object model/distributed component object model), one or more Oracle Call Interfaces (OCI), one or more Oracle Pro*C Libraries, and one or more Informix ESQL C Libraries.

Non limiting examples of an extension framework used as part or all of a package (asset) boundary include: one or more Unix Shell Scripts, one or more Windows batch files, a CICS Job Control Language, a Microsoft Visual Basis Script, and a Visual Basic.

Many of the boundaries of the sub application (package) are determined by the boundaries of one or more assets (240 below) contained in the sub application (package). In other words, the "boundary" of the sub application is determined by the collective boundaries of one or more of these assets contained in the sub application. The boundaries of these assets (asset boundaries) are defined the same way as the boundaries of the sub applications above.

There are many types of assets (also referred to herein as digital assets). Each of these assets has an asset type 240T. These assets also have intrinsic properties 240I (see below). Some examples follow.

A static content (SC) asset is a distinct file that is transferred from a HTTP server (e.g. a web server) to an HTTP client (e.g., a browser). The boundaries of the static content asset are the HTTP protocols that are necessary to move the SC asset from the HTTP server to the HTTP client. These boundaries define the asset as an SC asset type. Examples of information contained in SC assets include any HTML page, images (e.g., JPEG), movies or animation, an audio file (e.g., an MP3 file). Examples of HTML page assets could be a personal or corporate home page, an information portal, a sales vehicle, a text and/or image document, a interface to a server application (e.g., on-line banking, e-mail, financial services, sales services, advertising), an audio file, etc.

A dynamic content (DC) asset includes such assets as a Java Servlets or Java Server Pages (JSP). A Java Servlet is a single java class file that produces an HTML response to an HTTP request. The boundaries of the Java Servlet are the boundaries defined by the Java Servlet API for servlet components that define the asset as a dynamic content asset type. Note that JSPs are XML files that are compiled at run time into servlets therefore are defined by the same boundaries (APIs). There is also optional J2EE deployment descriptor information that is associated with these servlets and JSPs. The Java Servlet API specification and the JSP specification describe these class files, XML files, and J2EE deployment description information. Examples of information contained in DC assets include JSPs, Servlets, Microsoft ASPs (active server page), PHP (Hypertext Preprocessor), CGI, Cold Fusion by Allaire, Corp., etc. These DC assets are used for or used to generate a personal or corporate home page, an information portal, a sales vehicle, a text and/or image (e.g., JPEG) document,

a interface to a server application (e.g., on-line banking, e-mail, financial services, sales services, advertising), an audio file, etc.

An Enterprise Java Bean (EJB), including session beans (SB) and entity beans (EB), is defined by a component boundary specified in the EJB specification that defines the asset as an EJB asset type. An EJB has home, remote, and implementation class files associated with each other. The EJB also has J2EE deployment descriptor information associated with home (skeletons), remote (stubs), and implementation class files. The EJB specification describes these class files and J2EE deployment descriptor information. Examples of EJB uses include: business logic such as work flow systems; pricing algorithms; inventory management system; purchasing systems; dynamic customer, inventory, and pricing data; and other e-business systems.

Relational data assets include reference data assets (RD) and entity data assets (ED) have boundaries defined by a relational database table schema associated with each respective relational data asset that define the asset as an RD or ED asset type. The data in the relational data asset may be a complete set or a sub set of the data in the relational database table at any given time. This data is the result of some predefined query, e.g. an SQL query. Examples of relational data asset applications include information for: inventory, pricing, product, customer, weather, raw materials, employees, and any other business or personal data.

A java class asset (JC) is defined by a boundary determined by the creator of the class that also defines the asset as a java class asset type. These JC can include any purpose or use for programs and/or data.

Non-java assets also exist. For example, any program or sub program can be an asset with its boundaries defined as the data, information, and/or control paths (edges) of the call graph of the program. Examples of these types of assets include: C++, Small Talk, Visual Basic, Perl, and programs and/or data written in any other programming language that can be compiled and/or interpreted.

Asset types can fall into categories. For example, presentation asset types include assets that are involved with the presentation of information to a user, e.g. at the target computer. Usually a presentation asset type category includes DC, JB, and SB but could include EB as well. A transaction component asset type category includes assets that operate on information to cause some sort of change to the information. Generally, the transaction component asset type category includes EB but can also include DC, JB, and SB if these assets are used for a transaction purpose. A relational data asset type category includes assets that contain and/or manage data (e.g. data bases) and typically include RD and ED. Other asset type categories are envisioned, as are other asset types that would fit into these categories because these assets relate to the purpose of the category.

The sub application (package) may or may not be capable of a complete execution without the rest of the application. After distribution, the sub application may execute on its own, may execute on its own and exchange information with the rest of the application or other sub

applications, or execute in conjunction with the application (or other sub applications of the application) that are executing at different locations (e.g., sources, targets, middle ware servers, proxy servers, etc.) on the network, i.e. in a distributed execution over the network.

Sub applications (packages) are also designated by type, or preferably by a set of types. The package type is defined by the type, or types, of the asset(s) (240 below), i.e. asset type 240T, contained in the respective sub application (package). An asset type is an identifier of an asset, typically determined by a boundary of the asset, as described above. The asset type of the asset also determines what information (e.g. extended environment) 220, below, that the asset(s), and hence the sub application, need to execute on any given remote target environment or target base environment 250, below.

In this disclosure, sub applications are more specifically defined in terms of packages. (See below). As stated above, a package includes one or more assets 240 and has a type, or types, (i.e., package type) defined by the types of the asset that the package contains. The description of the package structure, including the package type(s), is called the package specification.

One novel feature of this disclosure is that packages are structures containing one or more assets 240 each of a particular asset type. In a preferred embodiment, these novel packages contain assets with one or more data asset types (e.g., a relational database asset type) along with one or more other assets that are not of a data base type. In an alternative embodiment, the novel packages contain one or more dynamic content (DC) asset type, with one or more assets that are not dynamic content (DC) asset types. In other embodiments, assets of different asset type category can be included in the same package. For example, an asset in the presentation category, e.g. DC or otherwise, or relational data category could be included with an asset not in those respective categories. In an alternative preferred embodiment, the some assets have a static content asset type and other assets have a presentation component asset type. Further, additional optional assets can have a relational data asset type and still other optional assets can have a transaction component asset type. Using features of this invention, these packages, even those with assets of dissimilar (category) asset types, novelly can be moved across the network to execute on any given target that contains a base environment (250 below) suitable to the package.

Figure 2 shows the general conceptual structure of an asset (also called digital asset) 240. As stated above, assets 240 have an asset type 240T determined by the asset boundary. Generally, the asset has two layers: the first layer is the Logic/Data (LD) layer 210, and the second layer is the Extended Environment (EE) layer 220. In order to establish context, an additional layer is shown; the Base Environment (BE) layer 250. An optional Asset Interface (AI) 230 is also shown between the EE 220 and the BE 250. In one preferred embodiment, the asset type determines what the minimum requirements are for the EE 220 so that the asset240 can perform an asset purpose or functionality (i.e., the purpose or functionality that the asset performed at the respective EIS) on any given BE 250 of any given target. In alternative embodiments (as described below) the EE 220 is a data structure containing

descriptors and/or other information required during one or more steps in the life cycle of the asset.

The Logic/Data layer 210 comprises any general type of program (algorithmic logic) and/or data that embodies the asset purpose or function.

In a more preferred embodiment, the Logic/Data (LD) layer 210 embodiments can include: sub systems, applications, sub applications, components, modules, functions, variables, and data in any language. In an even more preferred embodiment, the Logic/Data layer is in an object-oriented language that can include: components, modules, functions, classes, methods, and data members. In a still more preferred embodiment, in the object-oriented language is a Java programming language where the Logic/Data layer can include: java archive files (JARs), Java packages, and Java classes. In a still more preferred embodiment, using the Java Enterprise Edition, Application Programming Interfaces (APIs) have a Logic/Data layer that can include: Java Server Pages (JSPs), Java Servlets, and Enterprise Java Beans (EJBs). Data 210 can include any type of data structure including that data stored in a relational database, an object-oriented database, serialized objects, hierarchical database, and/or flat file. The Logic/Data layer can also be any combination of these things, including logic and no data, logic and data, or data and no logic. In a preferred embodiment, the Logic/Data layer is a subset of one or more EIS.

In one embodiment, the extended environment (EE) layer 220 is also a subset of the respective EIS but includes those portions of the EIS that are necessary to support a respective Logic/Data layer 210. The content of the extended environment layer 220 therefore depends on what the respective Logic/Data layer 210 is and what the asset type 240T of the respective Logic/Data layer 210 is. Support of the respective Logic/Data layer 210 enables the respective Logic/Data layer to perform the asset purpose in a different environment and/or on a different platform, e.g., on any given target computer. That is, in a preferred embodiment, the support provided by the EE layer 220 is that part of the source, e.g., EIS system, that is necessary for the respective LD layer 210 to perform the asset purpose on a remote target computer. An in a further embodiment, it might be necessary to transform the EE due to differences between the source and target environment. In yet another embodiment, the EE might be transformed into a neutral format that is not particular to any source or target environment, but, able to be more readily transformed than the source EE. In other preferred embodiments (as described below), the EE is a data structure containing one or more descriptors required during one or steps of the asset 240 life cycle 240L.

The neutral format is a format between a proprietary format of a source and a proprietary format of a target for a given asset type. For example for an EJB would be in the neutral format when the EE of the asset has neither the proprietary parts of the source environment or the proprietary parts of the target environment. For instance, J2EE deployment descriptors have proprietary sections that may be vendor specific yet still have other sections with a standard (e.g. publicly defined) representation, as well.

For example, if the Logic/Data (LD) layer 210 is an EJB, the extended environment layer 220

comprises the proxy stubs and skeletons, J2EE deployment descriptors, DataSource references, and JNDI entries associated with the respective EJB. If the Logic/Data layer 210 is data, the extended environment layer 220 typically comprises relational database data. See below for more detail.

In some preferred embodiments, there is an asset interface layer (AI) 230 between the EE layer 220 and the base environment layer (BE) 250 below. The AI 230 enables the passing of information, if required, between the EE 220 and the BE 250. The AI provides a standard interface between the asset adapter and the BE. In a preferred embodiment, the AI is a common interface that is available to the asset on any system supported by the DIS technology. There is an AI associated with each unique BE. There is an AI for each BE that corresponds to each asset adapter.

In a preferred embodiment, for each given BE 250 there exists an extended environment (EE) for each type of asset (asset type 240T) in that base environment 250. Also, for those assets requiring an AI 230, for each given BE 250 there exists a AI 230 for each asset type in that BE 250.

For the purposes of this disclosure, one preferred structure of an asset 240 includes: the LD 210, the EE 220, and, where applicable, the optional AI 230. The AI can be optional because it may be the case that the asset attaches directly to (e.g. communicates directly with) the BE.

The base environment layer 250 is not part of the asset, but is used to enable the asset in one or more environment and/or platforms (target system) other than those originally used on the EIS (source system). Enabling the asset 240 means providing that base environment layer 250 necessary so that the asset can perform the asset purpose or function on the remote target environment/platform. The base environment 250 might include web tier services, middleware component services, and/or database services. For instance, in some embodiments, the base environment layer 250 provides all or part of: a Java Virtual Machine, a web server engine, a Java servlet engine, an EJB container, a database management system (DBMS), and/or relational database management system (RDBMS).

An asset 240 also can be defined as some meaningful partitioning (logic/data layer 210) of an application from the source environment combined with part of the source environment (the extended environment layer 220) that is needed to run that partition of the application. Thus, in this case, both the LD 210 and the EE 220 are parts of the source environment (e.g. an EIS) that are selected so that the asset 240 can be moved across the network, particularly across network tiers, and so that the asset 240 can perform the asset purpose or function (the purpose or function that the asset/LD 210 was designed to perform on its respective source environment) on any given target, e.g. remote, computer environment and/or platform.

The asset type 240T of each asset 240 uniquely identifies a particular class of asset. In this way, assets can be correlated with the proper asset adapter(s) (see below) as the assets 240 move across the network, e.g., across tiers of the network, during the lifecycle 240L of the asset 240. See description below.

An asset adapter is a logical designation for a set of functionalities that enables the asset to progress through the asset lifecycle 240L. The asset lifecycle 240L is a set of transformations of the asset 240 as it moves from the source environment (e.g. one or more EIS), into the distribution environment (e.g. one or more distribution servers), to one or more target environments (e.g. one or more clients/nodes), and optionally back to the source environment (possibly back through the distribution environment.) These transformations are caused by several asset adapters and depend on the type of the asset 240. (See below for further description of asset adapters.)

A single asset 240 contains all the elements, such as data or information, necessary to package, move, transport, and restore the asset to its original state (that state in its source location) while and/or after moving the asset from the source location to the one or more target locations (and optionally back) so that the asset can perform the asset purpose or function on the target. Thus an asset 240 is able to maintain its relationship with the environment in which the asset is expected to perform the asset purpose or function across different environments or platforms. In a preferred embodiment, the asset 240 has the minimal application logic and extended environment (e.g., execution/transaction/security context and state) required to perform the asset purpose or function on one or more targets.

As stated above, the asset has an asset boundary to define the asset, as is the example case of a Java bean or EJB. Additionally, the asset may represent the state behind a well known functional interface, such as data that would be accessed through a standard set of calls, e.g., JDBC interface calls. If the JDBC interface is viewed as a discrete set of objects, the state corresponds to the relational data in the data source (e.g. a relational database).

The following table shows non-limiting examples of the different parts of an asset for different non-limiting example asset types 240T. The algorithmic logic, data, 210 and extended environment 220 are part of the asset itself. The base environment (BE) 250 might be expected to be provided for the asset but might not be required. Typically, the BE 250 is either known to be on a target computer, or to be provided on the target computer, before the asset 240 is required to perform the asset purpose or function.

In some preferred embodiments, the BE 250 itself might be sent to the target computer as one or more assets 240.

| Examples of asset types and their constituent parts. | | | | |
|---|---|---|---|---|
| | **Asset Type** | **Algorithmic Logic & Data** | **Extended Environment** | **Base Environment** |
| **Base** | CDS Adapter | Adapter Class Files | Version Information<br>Licensing Information | CDS Client Agent<br>JVM |
| **Java** | Static Content | HTML Page | Web Server Aliases | Web Server |
| | Static Content | GIF Image | | Web Server |
| | JSP | JSP Page | JNDI Entries | JVM<br>Servlet Engine |
| | Java Class File | Class File | JNDI Entries | JVM |

| | | | | |
|---|---|---|---|---|
| | Session Bean | EJB Implementation | EJB Stub and Skeleton<br>Deployment Descriptor<br>JNDI Entries | JVM<br>EJB Application Server |
| | Entity Bean | EJB Implementation<br>Data based on query | EJB Stub and Skeleton<br>Deployment Descriptor<br>JNDI Entries<br>Table Schema | JVM<br>EJB Application Server<br>DBMS |
| | Reference Data | Data based on query | Table Schema | DBMS |
| | Entity Data | Data based on query | Table Schema | DBMS |
| **Non- Java** | SmallTalk | Image | Image | SmallTalk VM |
| | C++ | Executable, or<br>Dynamic Library | Registry Entries<br>Environment Variables | Operating System or<br>Platform Emulator |
| | Perl | Perl Script | Environment Variables | Perl Interpreter |
| **Non-Language** | Music | MP3 File | Album and track information. | MP3 Player |
| | Video | MPEG File | Production Notes | MPEG Player |
| | Documents | PDF File | Authentication Certificate | PDF Viewer |

Assets 240 comprise many types (asset types) 240T, including: static content (SC), dynamic content (DC), java beans (JB, session bean (SB), entity bean (EB), reference data (RD), entity data (ED), naming directory, and many more. (Java beans, session beans, and entity beans are trademarks of the Sun Microsystem Corporation.)

Static content assets (SC) include information that doesn't change in a program and/or display. Static content assets can be cached in a local, e.g. client memory, for re-use so that the asset doesn't have to be resent over the network each time it is used. Examples of static content assets include html files, gif files, and jpeg files (jpg).

Dynamic content assets (DC) include information that changes with time. Often this information is displayed/provided with SC. Typically, the DC is produced in real-time or at the time of use. For example, a weather map of the United States might be locally cached as SC but the temperature in New York City is DC that is continually updated in time as numeric data that is displayed on the map near the location of NYC. In a financial application, forms or format for accounts (e.g. loan applications, mortgage applications, stock/financial instrument portfolios, bank statements, a service or product advertisement, etc.) are provided as SC while the offerings and/or value of particular financial assets is provided as changing DC (e.g., interest rate, stock price, an account balance, or a cost of service or product). The DC could be a presentation asset type category, a transactional component asset type category, and/or another asset type category depending on the function performed by the DC.

Java Beans (JB) is a well known Java class file that follows the convention for JavaBeans. The convention specifies how to name the methods in order for third party tools to provide automated processes for the class. In this disclosure, "Java Beans" (JB) are used sometimes to indicate a general Java Class. The EB could be a presentation asset type category, a transactional component asset type category, and/or another asset type category depending on the function performed by the EB.

A Session Bean (SB) is a well-known Enterprise Java Bean that is intended to be coupled with a client session. The SB is well defined in many books and the EJB specification. The SB

could be a presentation asset type category, a transactional component asset type category, and/or another asset type category depending on the function performed by the SB.

An Entity Bean (EB) is an Enterprise Java Bean that is intended to represent a unique piece of data that can only be accessed by one client at a time. The EB is well defined in many books and the EJB specification. The EB could be a presentation asset type category, a transactional component asset type category, and/or another asset type category depending on the function performed by the EB.

The term Reference Data (RD) is a term we defined to refer to a set of database records that are intended to be accessed in a read-only manner. These records are intended for "reference" purposes, and are sent to the client or an intermediate target in order to be accessed as an application executes. The same RD can be sent to several targets without worrying about the overlap in data that different targets have. This safety is due to the fact that the data will not change.

The term "Entity Data" (ED) describes data that is like the RD except that the ED is expected to change. The ED asset type must be treated differently because of the problems inherent in the synchronization of data that has been replicated to different targets. It is not apparent how changes in the same data should be synchronized on the back-end database from several targets. For example, do you allow target 1 to update a record, then allow target 2 to update the same record, even when target 2 did not have the changes made by target 1 when it made its update on the remote node? So, by creating a distinct asset type, we are able to enforce the integrity of the data as it moves through the system. See below for a more detailed description of synchronization.

Both the ED and RD are relational data, e.g. in the relational data asset type category. One novel feature of these packages used in some preferred embodiments is that the packages contain both assets of a relational data type and assets that are not of a relational data type. The present disclosure shows how to identify and process different asset types in any general package that is moved over one or more networks (tiers of a network). In this way, all asset types in a sub application (package) can be used together.

In some preferred embodiments, one or more RD's or ED's (or both) are combined with any combination of SB, EB, and JSP.

As stated above, a package structure can contain one or more assets of a first asset type or category of asset type and one or more assets of a second asset types that are not of the first type or category. Assets with third, forth, etc. asset types or category of asset types can also be included in the package structure. For example, in a preferred embodiment, a package would include one or more (first) assets from the relational data asset category (e.g., ED and/or RD) and one or more (second) assets from the presentational asset type category (e.g., DC, JB, SB, and/or EB). One or more third assets from the transactional component asset type category might be added, e.g. DC, JB, SB, and/or EB. One or more static content asset types (SC) might be added to the package or replace the transactional component asset type category

asset.

In an alternative preferred embodiment, the package structure can be a combination of first assets with a static content asset type and second assets have a presentation component asset type (e.g., DC, JB, SB, and/or EB). Further, the optional third asset can have a relational data asset type (e.g., ED and/or RD) and the optional forth asset can have a transaction component asset type (e.g., DC, JB, SB, and/or EB).

Note that in a preferred embodiment, the presentation component asset type is used to generate content and/or perform algorithmic logic associated with the manipulation and/or validation of user interface data and/or content. Content may include: HTML, Gif files, JPEG, WML, and XML, and any other general markup language. In a more preferred embodiment, the presentation component asset type is a dynamic content asset.

Digital assets 240 have intrinsic properties 240 I. The properties are Volatile 280C, Runnable 282C, Non-Volatile 284C, and Non-Runnable 286C. See Figure 2C, which is a UML (Unified Modeling Language) diagram of an Asset Class.

A Volatile Digital Asset 280C can be universally distinguished in any computing environment because two successive reads of the digital asset might return different results. In distributed computer application environments (such as a client/server application), a Volatile Digital Asset can be further distinguished in that working copies of the Volatile Digital Asset (typically located on a client computer) need be reconciled with the master copies at the source (typically an Enterprise Information System) only at prescribed synchronization points.

A Runnable Digital Asset 282C can be universally distinguished in any computing environment because instances of this Digital Asset are intended to be and are capable of being executed by any operating system thread.

A Non-Volatile Digital Asset 284C (herein also referred to as a Static Content Asset (SC) or alternatively, as a Static Asset), can be universally distinguished in any computing environment because there exists exactly one view of the asset, no matter how many instances of the asset may eventually be created. A Static Asset appears and is immutable regardless of its location in any network.

A Non-Runnable Digital Asset 286C can be universally distinguished in any computing system because instances of this Digital Asset are not intended to be, and are not capable of being executed by any operating system thread.

The properties of Runnable, Volatile, Non-Volatile, and Non-Runnable are well known in the prior art, for example, see:

Java Language Specification, 2nd Edition, Draft Gosling, et. al., Copyright 2000 by Sun Microsystems, Inc., Page 165.

C++ Programming Language, 3rd Edition,Bjarne Stroustrup, Copyright 1997 by AT&T, Page 808.

Which are herein incorporated by reference in their entirety.

A whole is an association of one or more digital assets 240. This association can be a physical association (e.g., where the whole is a well known container of digital assets 240) or a logical association (e.g., where the whole is a system 105F, sub-system 106F, application 108, sub-application 110F, components 120F, modules 130F, or a function 140F).

Certain of the intrinsic properties 240I of digital assets 240 are transitive to any whole of which the respective digital assets 240 are members. Specifically, if a whole contains or has a composition of one or more runnable digital assets 284C, the entire whole has a runnable intrinsic property. If the whole contains or has a composition of one or more volatile digital assets 240, the entire whole has a volatile intrinsic property. However, all of the digital assets 240 contained in the whole must be static 282C for the whole to have an intrinsic static property. Similarly, all of the digital assets contained in the whole must be non-runnable for the whole to have an intrinsic property of non-runnable.

The whole can contain or have a composition of digital assets 240 that are either homogeneous or heterogeneous. Here homogeneous or heterogeneous is with respect to the intrinsic property of the digital assets contained in or being members (composition) of the whole. Examples include homogeneous runnable wholes or a heterogeneous runnable wholes. Runnable wholes can be homogenous or heterogeneous.

Association can be a physical association or a logical association or both. A physical association can be defined from any grouping of two or more digital assets, e.g. a file containing two or more records (objects, structures, etc.) is a physical association. A logical association of digital assets 240 describes discrete connections among the respective digital assets 240 in the respective whole. A logical association carries information about relationships among the digital assets in the whole, e.g. two or more runnable objects logically associated by a call graph. Another example of a logical association is the HTML code of a web page that identifies and logically associated a number of executable digital assets 240 (e.g., Java Scripts) and one or more static content component (e.g. a wav file). A logical association can also be define by one or more rules, e.g. specifications of the association. These rules could describe group/role associations; business, marketing, and/or pricing associations; or any general association that can be specified or defined in this way.

An example of a logical and a physical association is a containment graph representing all executable and data files in an application 108 and their topology that may physically reside in a single data structure in memory, such as a package (see below). Another example of a logical and physical association is a collection object such as a vector collection object in C++, which may store one or more homogeneous digital assets 240 in memory, with the basis of their joint association preserved externally from the collection. Here it becomes apparent that logical associations can be internal (within the whole) or external (external to the whole).

A digital asset 240 shares a joint membership with other associated digital assets 240 in a whole. Generally, if the whole contains at least one runnable digital asset, this whole is any of the following parts 100F: a system 105F, a sub-system 106F, application 108, a sub-application 110F, a component 120F, a module 130F, or a function 140F. (See the description of Figure 1F.) Typically, whole that contain at least one runnable asset also have other associated digital assets associated with the runnable digital asset. Often these wholes contain non-runnable digital assets as well so the whole is heterogeneous.

A composition is an assembly of parts that forms a whole. The composition part-whole relationship is well known, and generally, the system parts 100F are well known instantiations of compositions.

Figure 2A shows a preferred description of an asset lifecycle 240L. The asset lifecycle is a description of the asset, and changes to the asset, as the asset moves across tiers of the network onto different computer platforms and environments.

The asset goes through a lifecycle 240L starting in the source tier, moving through the deployment tier, into the target tier, and then optionally back through the deployment tier to the source tier and/or can move to any other node or nodes in the network if required. The asset adapter methods are used in one or more of the steps in this lifecycle. In one embodiment, an asset type specific processing is required for the asset to continue through the lifecycle. In some preferred embodiments, information contained in the EE 220 is modified by one or more of these adapters to enable the asset to continue through the life cycle.

In a preferred embodiment, assets 240 are "discovered" in the source environment (tier) 910 by an asset adapter called the discovery asset adapter method 2100 (see Figure 21 below). In alternative embodiments (Figure 21B) the discovery asset adapter method identifies and creates the digital asset 240, creates an intermediate representation 2100C of a computer system part 100F, adds the asset to an Asset Inventory 2100D, and writes asset descriptors into the EE 220. Optionally, an asset description data structure 1170 is created.

In a preferred embodiment, an "export asset adapter method" (see Figure 16) identifies and exports candidates for classification as assets 240 and together as packages. In this preferred embodiment, the export asset adapter method 1600 (see Figure 16 below) is used to obtain the actual current version of assets in the source tier that needs to be distributed to the target tier. After the assets are exported, the assets are moved to the deployment or distribution tier 960 and typically stored in an asset cache 961. When exporting assets, the export asset adapter method captures the logic, data, 210 and extended environment information 220 for an asset 240 and puts it into an asset data structure 240 where the asset type 240T is also identified.

In another preferred embodiment, the export asset adapter method (see Figure 16B) takes the Asset Inventory 2100D and the intermediate representation 2100C and prepares a preliminary package specification 1650B. The asset data structure and EE 220 of the digital asset 240 are also updated.

A package specification 1100 (see Figure 4 below) is created that in turn contains asset specifications 1170. In a preferred embodiment, the asset specification 1170 is stored in the deployment tier until a package 1100 is scheduled to be delivered.

In an alternative embodiment, a packaging agent takes the preliminary package specification 1100A and creates a finalized package specification data structure. In a preferred embodiment, both the preliminary package specification 1100A and the finalized package specification 1100A have the same data structure, comprising an Extended Environment-Package data structure 1120A and one or more Asset Identifiers 1172 (see Figure 11A).

The optional version asset adapter method 1660 (see Figure 16A below) is used to determine the current version information of the assets 240 in the source tier. In a preferred embodiment, this version information is compared with the target tier asset version information in the deployment tier to determine if assets need to be deployed from the source tier to the target tier.

If an asset requires processing the processing may be done when the asset is stored in the asset cache 961 or at any time before the asset is distributed to either a secondary cache in the deployment/distribution tier 960 or the target tier. The processing is primarily performed on the asset's extended environment 220, in an attempt to translate the extended environment 220 to run in harmony with the base environment 250 in the target tier. However, the processing process (see Figure 18 below) may also change the logic/data portion of the asset or both the logic/data and the extended environment portion of the asset.

An agent 1400 in the target environment requests the assets that are pending in the CDS/ADS cache for delivery to the target tier are in fact delivered to the respective target.

In a preferred embodiment, the target processing asset adapter method 1900 (see Figure 19 below) is executed on the CDS/ADS tier against any asset 240, (typically cached in the CDS/ADS cache) that requires targeted processing before being sent to the target tier 935. Target processing is intended primarily to change the Logic/Data section 210 of the asset data structure 240 in order to provide a unique asset that can create or has personalized information for one or more specific targets (typically 940) on the target tier 935 on which the asset 240 is being deployed. The processed asset 240P therefore can have a changed Logic/Data section 210P. However, the processed asset 240P could have a changed extended environment section 220P or both parts (210P and 220P) can be changed. In other cases, neither the Logic/Data section (210, 210P) nor the extended environment (220, 220P) will be changed. The processed asset 240P is typically stored in the CDS/ADS cache 961.

In some embodiments, a targeting process adapter 1900 can target to an intermediate target (a server that will in turn server many users) or a final target (a single node that will serve a single user).

When the asset is sent to the target tier, the deploy asset adapter method 1700 (see Figure 17 below) is invoked to deploy the asset (240, 240P) into the computational environment, i.e., the base environment 250, in the target tier. The extended environment 220P from the asset's data structure 240P is used to set the base environment 250 and extended environment 220P in the target tier to run the asset 240P in a correct manner. The asset's logic and data 210 are then deployed into the base environment 250, and since the environment has been adjusted, the logic 210 will function correctly and the data 210 will be accessible.

When changes happen in the target tier 935 that warrant synchronization, the synchronization asset adapter method 2000 (see Figure 20 below) is executed to create a synchronization asset 240S that is propagated back through the deployment/distribution tier 960 (optionally) and then into the source tier 910. The source tier resource (asset) that corresponds to the synchronization asset is synchronized with the information in the synchronization asset 240S.

Figure 2B is a block diagram of a preferred extended environment (EE) 220 of any general digital asset 240. The extended environment data structure 220 is part of the digital asset 240 that is capable of being transmitted over one or more multi-tiered networks. A preferred embodiment of the EE 220 is written in XML (extended markup language). An example of a preferred EE 220 written in XML is given below with associate reference numbers that refer to Figure 2B.

The EE 220 has one or more common descriptors 210B, one or more asset dependency descriptors 222B, and one or more target server dependencies 226B. In an alternative preferred embodiment, the EE 220 additionally has one or more EIS server dependencies 224B. In other preferred embodiments, additional descriptors are added as described below.

The common descriptors 210B provide a unique identification of the digital asset 240 on the networks. Examples of common descriptors 210B include any one or more of the following: a digital asset name of the digital asset, a unique fully qualified name of the digital asset, an address of the digital asset, a size of the digital asset, a volatility descriptor of the digital asset, a common runnable descriptor of the digital asset, a user type descriptor of the digital asset, a security descriptor of the digital asset, and a price descriptor of the digital asset.

The asset dependency descriptors 222B identify one or more associated digital assets 240. The associated digital assets are those digital assets 240 that are associated with the digital asset 240 that contains the subject EE 220. In a preferred embodiment, this association defines a joint membership of the digit asset 240 and the associated digital assets as parts of a whole. See above.

Examples of the asset dependency descriptors 222B include any one or more of the following: one or more names of the other digital assets on which the respective digital asset is dependent; any general asset identifier, and/or one or more unique fully qualified names of other digital assets on which the digital asset is dependent.

The associate digital assets 240 and the digit asset 240 have joint membership in a whole that defines a part-whole relationship. In a preferred embodiment, the whole is defined by a graph. Graphs are well known in the computer arts and define physical and/or logical associations among the digital assets 240 in the whole. In a more specific preferred embodiment, the graph is any one or more of the following: a containment graph, a tube graph, a call graph, and a pure representation expressible as a graph.

In some preferred embodiments, the whole is embodied as a package. This is a physical and/or logical association. In a preferred embodiment, the EE 220 has package relationship descriptors 285B that represents a part-whole relationship between the digital asset 240 and one or more packages containing the digital asset.

In some embodiments, the package relationship descriptors represent at least the following three relationships in the part-whole relationship: a mandatory part-whole relationship, a shared part-whole relationship, and a root part-whole relationship.

A mandatory part-whole relationship is defined here as an invariant relationship between a specific digital asset and a specific package. That is, the specific package does not possess the property of correctness unless the specific digital asset is included. In an alternative embodiment, this relationship can further imply the existence of the other digital assets that are members of the specific package at a given location on the network when the specific digital asset is established to exist at that location.

A shared part-whole relationship is defined here, in a preferred embodiment, as a component-integral relationship between a specific digital asset and one or more specific packages. That is, the specific digital asset may be included in one or more packages. The component-integral relationship indicates that the digital asset bears a particular functional or structural relationship with each of the one or more specific packages. The shared relationship descriptor indicates the component-integral form of the relationship. In an alternative embodiment, the shared relationship descriptor can indicate that the specific digital asset has a member-collection relationship with one or more specific packages. That is, the digital asset may be included in a collection of digital assets comprising one or more specific packages.

A root part-whole relationship is defined here as a non-mandatory "base" or "foundation" relationship between a specific digital asset and one or more specific packages of varying composition.

See Winston, M.E., et al., "A Taxonomy of Part-Whole Relations", Cognitive Science, 11, 1987, pp. 417-444, which is herein incorporated in its entirety, and other well known works on this topic.

The EE 220 includes one or more base environment descriptors 225B (e.g., target server dependencies descriptors 226B) that identify a base execution environment on one or more target computers. The base execution environment 250 is required to execute the digital asset

240 on any given target computer 830 to which the digital asset 240 is distributed. The base execution environment 250 has zero or more other digital assets 240. In one embodiment, one or more of the other digital assets 240 are deployed to the respective target computer from one or more or the Enterprise Information Systems (EIS) 810 to create the base environment 250. In one preferred embodiment, the minimum complement of other digital assets required to utilize the respective digital asset at the target are deployed to create the base environment on the target.

In a preferred embodiment, the digital assets deployed to the target computer from the EIS to create the minimum base environment are named as the target server dependencies in the EE 220. These target server dependencies can include any one or more of the following: one or more database management systems (DBMS), one or more browsers, one or more Java Virtual Machine (JVM) instantiations, one or more operating systems, and other systems 105F, sub-systems 106F, applications 108, sub-applications 110F, components 120F, modules 130F, and/or functions 140F. When such system part is named in one of the target server dependencies descriptors 226B for the digital asset, the packaging process (see below) will perform a lookup of a pre-prepared manifest for that respective system part and incorporate digital assets to deploy the respective system part 100F.

In an alternative embodiment, the EE 220 has one or more EIS server dependencies descriptors 224B. The EIS server dependencies descriptors 224B identify an EIS execution environment required on the EIS 810 in order to ensure correct utilization of the digital asset on the EIS. The EIS execution environment is that set of EIS systems 105F, EIS sub-systems 106F, EIS applications 108, EIS sub-applications 110F, EIS components 120F, EIS modules 130F, and EIS functions 140F that the digital asset 240 requires on the EIS 810 in order to ensure correct utilization of the digital asset on the EIS. The EIS server dependencies descriptors 224B can be used to ensure that digital assets that reside on the EIS will be correctly synchronized (see synchronization below) with digital assets deployed to one or more of the target systems.

Examples of EIS server dependencies 224B include one or more of the following: EIS operating systems (in rare cases), EIS database management systems (DBMS), EIS servers, EIS application servers, EIS web application servers, one or more accounting, customer relationship management (CRM) systems, business to business (B2B) systems (e.g., supply chain management, etc.), business to customer (B2C) system (e.g. order fulfillment systems, electronic shopping systems, etc.), and one or more message oriented middleware applications. Specific EIS server dependencies 224B include one or more of the following: one or more Oracle DBMS, one or more Sybase DBMS, and one or more DB2 DBMS.

Further, one or more of the target server dependencies and one or more of the EIS server dependencies can be compared to determine whether a transform of the digital asset is required for the asset to be deployed on the respective target computer (see below).

In one preferred embodiment, the EE 220 has one or more transform descriptors 255B that facilitate and/or enable a transform of the digital asset 240 from its form in the EIS execution

environment to a form utilizable in the base execution environment 250. In a preferred embodiment, the transform descriptors 255B may include a properties descriptor, a format descriptor, and a registry descriptor (e.g. primarily for Window 32 systems).

The properties descriptor that provides information required for use of the digital asset on an operating system of the base execution environment 250. For example, the Sun Microsystem EJB API call, EJBContext.getEnvironment() method is optional in EJB version 1.1, which means that it may or may not be supported in a different base execution environment (i.e. a base execution environment supporting EJB 1.0). In this example, a properties descriptor could indicate an action to take (typically in the form of an exception) if the digital asset was required to execute in an EJB 1.0 base execution environment. In this example embodiment, the values in the properties descriptor would be used to modify the base execution EJB 1.0 environment to handle this API call.

The format descriptor provides information required for use of the digital asset on an operating system of the base execution environment 250. For example, in order to perform one of the example transforms described below, that is the transform of a Unix plain text file to a form usable in a MS-DOS or Windows system, a key-word, such as UNIX-TEXT would be set as the format descriptor, identifying the file as being in Unix text format.

The registry descriptor provides information required for the digital asset that is being deployed from (to) a non-Windows EIS (base execution environment) to (from) a Window's environment on the base execution environment (Windows EIS environment). A "registry" is a Window's 32 operating system feature that is used by Windows to manage hardware and software under its control. The registry is presented, and can be manipulated as a list of registry entries (e.g., text, etc.). Many Window's programs require specific entries be inserted into the registry for correct program operation.

The transformation of the digital asset could be a transformation of data in a logic/data section 210 of the digital asset 240. For example, if the digital asset 240 is a standard text file (see transform 255B format descriptors above), and if the EIS server dependency descriptors 224B indicate the EIS has a UNIX environment, and if the target server dependencies descriptors 226B indicate that the target has a DOS environment, then, the transformation would include insertion of the "^M" character into the LD section 210 at the end of each line. This transformation can be initiated by a rule base system that has rules that apply to these conditions. See below.

The transformation of the digital asset could be a transformation of logic in a logic/data section 210 of the digital asset 240.

For example, one method to transform logic in a logic/data section 210 of the digital asset 240 is to send the Java source code statements for the logic digital asset to the target environment (or, alternatively on the CDS, DIS, below). In order to execute the Java language statements of a Java language program, the statements are interpreted and executed by a Java Runtime Compiler specific to a particular execution environment, e.g. an EIS environment or a target

environment. If the common runnable descriptor is set (indicating that the digital asset 240 is runnable), and if the runnable descriptor 240B (below) are set to values that indicate that the EIS and target environments are different (for example, Solaris and Windows), then a transform is effected by means of the two Java Runtime Compiler instances, respectively on the EIS and target environment. The runnable object created by the Java Runtime Compiler on the EIS is not the same as the runnable object created by the Java Runtime Compiler on the target.

In another example, the Sun Microsystem EJB API call, EJBContext.getEnvironment() method is optional in EJB version 1.1, which means that it may or may not be supported in a different base execution environment (i.e. a base execution environment supporting EJB 1.0). In this example, the properties descriptor is used as described above to realize the transform of the logic digital asset from a form that could only run on EJB version 1.1 to a form that could run on EJB version 1.0.

In an alternative preferred embodiment, the EE 220 also has one or more reference descriptors 260B. In a preferred embodiment, the reference descriptors 260B include any one or more of the following: a reference link descriptor, a reference file descriptor, and a reference directory descriptor. The reference link descriptor provides a world-wide-web (WWW) address that has contents used for processing of the digital asset. The reference link descriptor provides a world-wide-web (WWW) address that has contents used during execution of the digital asset. The reference file descriptor is a unique fully qualified name of a file required for reference by the digital asset. The reference directory descriptor provides an additional address information that is used to locate one or more of the associated digital assets. For example, the additional address information could include root or parent level directory names, which would, following well known format, be combined with the leaf directory level identification given in the common name descriptor to locate and fully qualify the path information for the digital asset.

In an alternative preferred environment, the EE 220 further comprises one or more asset type descriptors 230B. As a non-limiting example, the asset type descriptors 230B may describe any one or more of the following asset types: static content (SC), dynamic content (DC), enterprise java beans (EJB), reference data (RD), session bean (SB), entity bean (EB), entity data (ED), java class (JC), and java beans (JB). See above.

In an alternative preferred environment, the asset type descriptors 230B can be one or more asset category descriptors. Example asset category descriptors may include any one or more of the following: a presentational descriptor, a transactional descriptor, and a relational data descriptor. See above.

In an alternative preferred embodiment, the asset type descriptors 230B can be one or more asset class descriptors. Example asset class descriptors may describe any one or more of the following: base, java, non-java, language, and non-language digital asset classes.

In an alternative preferred embodiment, the EE 220 further comprises one or more package relationship descriptors 285B that represent a part-whole relationship between the digital asset 240 and one or more packages (see below) containing the digital asset. The package relationship descriptors 285B represent at least the following three relationships in the part-whole relationship: a mandatory part-whole relationship, a shared part-whole relationship, and a root part-whole relationship.

In an alternative preferred embodiment, the EE 220 further comprises one or more security descriptors 280B. The security descriptors are well known and may describe any one or more of the following functions: encryption, authorization, and access control.

In an alternative preferred embodiment, the EE 220 further comprises one or more runnable descriptors 240B. The runnable descriptors need only include a target execution environment for the digital asset 240. In an alternative embodiment, the runnable descriptors 240B may include an EIS execution environment and a target execution environment for the digital asset 240. In an alternate embodiment, as described above, if both the EIS execution environment runnable descriptor and the target execution environment runnable descriptor are both set, and their values are different, the transform process may be invoked on the runnable digital asset.

In an alternative preferred embodiment, the EE 220 further comprises one or more non-runnable descriptors 242B. The non-runnable descriptors need only include a target execution environment for the digital asset 240. In an alternative embodiment, the non-runnable descriptors 242B may include a EIS execution environment and a target execution environment for the digital asset 240. In an alternate embodiment, as described above, if both the EIS execution environment non-runnable descriptor and the target execution environment non-runnable descriptor are both set, and their values are different, the transform process may be invoked on the non-runnable digital asset.

In an alternative embodiment, the EE 220 further comprises one or more personalization descriptors that enable the digital asset to be customized upon delivery to one or more of the base execution environments. In a preferred embodiment, the personalization descriptors include one or more data keys that establish a linkage among data elements in the EIS execution environment. Alternatively, the personalization descriptors include one or more data keys that establish a linkage among logic elements in the EIS execution environment.

In an alternative embodiment, the EE 220 further comprises or more pricing descriptors 275B. The pricing descriptors describe information about any well-known general pricing information including one or more of the following: a price, a price scheme (subscription, pay to own, pay to use, one time payment), a payment detail, and payment method (check, credit card, card number).

In an alternative embodiment, the EE 220 further comprises one or more target information descriptors 270B. Target information descriptors 270B can give any general information about the targets and, by example, may include any one or more of the following: a well known user, an anonymous user, one or more user groups, an entire user group, a target

machine, an identifiable segment of target machines, a collection of target machines, an internet protocol address mask, and a group of target computers in a node collection structure.

In an alternative embodiment, the EE 220 further comprising one or more schema descriptors 250B. The schema descriptors provide information that describes any or more of the following examples: database table names and definitions, database column names and definitions, database key identifiers and value ranges, database view names and definitions, and other well known database schema elements.

In an alternative embodiment, the EE 220 further comprises one or more metadata descriptors 250B. The metadata descriptors 250B provide information that describe any or more of the following examples: repository object definitions, scope object definitions, module object definitions, operation object definitions, exception object definitions, constant object definitions, properties object definitions, attribute object definitions, relationship object definitions, type object definitions, and other well known metadata object definitions.

In one preferred embodiment, the EE 220 further comprises one or more distribution logic descriptors 290B. The distribution logic descriptors 290B describe or point to one or more transaction rules and one or more concurrency rules. The transactions rules specify any of a number and a frequency of times that the digital asset can be distributed to one or more target computers. The concurrency rules specify whether or not there are any restrictions on distribution of the digital asset with respect to the distribution of one or more other digital assets.

For an example of a transaction rule, a well known practice in the distribution of priced digital assets is that buyers of such assets are permitted up to some number of download attempts in the course of realizing internet-based delivery of a priced digital asset. This practice has been put in place to protect buyers from internet infrastructure failures that result in a failure of the buyer to receive all of the digital assets the buyer has purchased. Purchases of digital assets are regarded as transactions in the well known sense. The "up to [number]" practice represents a transaction rule to govern that purchase.

For an example of a concurrency rule, a well known practice in the internet-based distribution of software programs is that a user is blocked from downloading a certain version (for example, a "Release 4" version) of a software program while simultaneously downloading an incompatible version (for example, a "Release 5" version) of a sub-component of such software program.

Other EE 220 descriptors are envisioned.

As the digital asset 240 is distributed over tiers of the network(s), the EE 220 can be sent over one or more network interfaces, received over one or more interfaces, and stored on one or more memories through out the network. Various processes (e.g., discover, export, process, target, etc. below) will operate on the EE 220 or use information in the EE to perform their respective functions.

The following is an example preferred embodiment of the EE defined in Figure 2B as an XML document:

```
<?xml version="1.0" encoding="ISO8859_1"?>

<!DOCTYPE dis-ee PUBLIC '-//International Interactive Commerce, LLC//DTD Extended Environment//EN' 'http://www.iic-ltd.com/dis/dtds/iic_ee_1_1.dtd'>

<extended_environment>
        <common> 210B
                <name></name>
                <address></address>
                <size></size>
                <volatile></volatile>
                <runnable></runnable>
                <version></version>
                <user_type></user_type>
                <security></security>
                <priced></priced>
        </common>
        <dependencies> 220B
                <asset_dependencies> 222B
                        <asset></asset>
                </asset_dependencies>
                <eis_server_dependencies> 224B
                        <dbms></dbms>
                </eis_server_dependencies>
                <base_environment> 225B
                        <target_server_dependencies> 226B
                                <dbms></dbms>
                                <browser></browser>
                                <jvm></jvm>
                        </target_server_dependencies>
                </base_environment>
        </dependencies>
        <type_descriptors> 230B
                <type></type>
                <category></category>
                <class></class>
        </type_descriptors>
        <runnable> 240B
                <eis_execution_environment></eis_execution_environment>
                <target_execution_environment></target_execution_environment>
        </runnable>
```

```
<non_runnable> 242B
        <eis_execution_environment></eis_execution_environment>
        <target_execution_environment></target_execution_environment>
</non_runnable>
<schema_and_metadata> 250B
        <database>
                <process_granularity_level></process_granularity_level>
                <object_granularity></object_granularity>
        </database>
        <metadata></metadata>
</schema_and_metadata>
<transform> 255B
        <properties></properties>
        <registry_entries></registry_entries>
        <format></format>
</transform>
<references> 260B
        <reference_links></reference_links>
        <reference_files></reference_files>
        <reference_directories></reference_directories>
</references>
<personalization_keys>      265B
</personalization_keys>
<target_info> 270B
        <target_type></target_type>
</target_info>
<pricing> 275B
        <actual_price></actual_price>
        <price_scheme></price_scheme>
        <payment_details>
                <payment_method></payment_method>
                <credit_card_info></credit_card_info>
        </payment_details>
<pricing>
<security> 280B
</security>
<package_relation> 285B
        <mandatory></mandatory>
        <shared></shared>
        <root></root>
</package_relation>
<distribution_logic> 290B
        <transaction_rule></transaction_rule>
        <concurrency_rule></concurrency_rule>
</distribution_logic>
```

</extended_environment>

Figure 2C is an UML diagram of one preferred embodiment of an asset class object. This UML diagram shows the inheritance hierarchy of the digital asset 240. In this Figure, the digital asset is the superclass from which two subclass are derived. These two classes are the class of volatile digital asset 280C and static digital asset 282C. In turn, the volatile digital asset class is the superclass of a runnnable asset class 284C and the static digital asset class is a superclass of a runnable static digital asset class 286C. When this class hierarchy is respected by process that operate on these classes, the well known benefit of object oriented design can be realized. The properties of volatile, static, runnable are intrinsic properties 240I of the respectively correspondingly name class as described above. Following well known principles objects that contain or are compositions of concrete instances of these class will assume the transitive intrinsic properties of these classes. Therefore, concrete data structures (e.g., packages below) can be created to have these same intrinsic properties.

Figure 3 is a block diagram of one preferred package content data structure 1190 that identifies which assets 240 are packaged in which packages (1110, 1050E). This data structure provides information concerning which assets 240 are packaged, and/or are to be packaged, together in one or more of the packages.

In this embodiment, each record, typically 1195, of the package content data structure 1190, contains an asset field 1194 associating one (or more) assets 240 (see also 1012 in Figure 10 as an example) with a package in a package field 1192. Typically, the package identification 1110 (see Figure 11) is stored in the package field 1192 and the asset identification 1172 (see Figure 4) is stored in the asset field 1194. In an alternative embodiment in which all packages are single assets, this table 1190 may be omitted.

In a preferred embodiment, the asset content of one or more of these packages, e.g., a package structure 1190, is defined by the combinations of assets contained within the package.

While the package structure 1190 is defined by Figure 3, any well known way of identifying assets within the package structure can be used. As stated above, some package structures have a novel combination of assets including one or more assets from the relational data category with one or more assets from the presentation component asset category. Optionally, assets from the transaction component asset category and/or the static content asset category can be included.

In another alternative novel embodiment, the package structure includes one or more assets from the static content asset category with one or more assets from the presentation asset category. Optionally, assets from the relational data category and/or the transaction component asset category can be included in this embodiment.

In a preferred embodiment, the transaction component asset category performs typical business logic functions and/or manipulation of data in relational databases. Examples of transactional components include: an EJB entity bean, EJB session beans, dynamic content that is does database access, and/or java class that has business logic and/or does database access.

Other novel combinations of assets in the package are envisioned, e.g., including one or more legacy system assets in the package. Some more examples of possible package structures follow:

A package structure 1190, e.g., one with any of the asset combinations described herein, can be a sub application of an application on one or more Enterprise Information Systems (EIS).

A package structure 1190, e.g., one with any of the asset combinations described herein, can be a non-proprietary sub application of a proprietary application on one or more Enterprise Information Systems (EIS). Alternatively, the package structure 1190 can be a smaller sub application of the EIS that can be run on a target system with less capability that the EIS. Alternative, the package structure 1190 can be any sub set of the EIS that the enterprises chooses to distribute over the network to be executed on a remote target.

One or more of the assets 240 in the package structure 1190 can be from the presentation asset category or transaction component asset category and have a logic/data part includes any one or more of the following: a sub system, an application, a sub application, a component, a module, a function, a variable, of an EIS program, and a data structure. In a preferred embodiment, this logic/data part is embodied in an object-oriented language that includes any one or more of the following: one or more components, modules, or functions, one or more object classes, one or more object methods, and one or more data members. In the case where this logic/data part is embodied in an object-oriented language the logic/data part can include any one or more of the following: one or more java archive files (JARs), one or more Java packages, and one or more Java classes. In a more preferred embodiment, this logic/date part is an Java Enterprise Edition programming language object-oriented language that includes any one or more of the following: one or more Java Server Pages (JSPs), one or more Java Servlets, and one or more Enterprise Java Beans (EJBs).

In an alternative preferred embodiment, the package structure 1190, e.g., one with any of the asset combinations described herein, has one or more assets types from a relational data category that may include elements in addition to the relational data. These element include any one or more of the following assets: any type of data structure, a set of relational database data, a set of object-oriented database data, one or more serialized objects, a set of hierarchical database data, a subset of EIS data, one or more data sets extracted from one or more XML structures, and a flat file.

In an alternative embodiment, the package structure 1190, e.g., one with any of the asset combinations described herein, has one or more assets with extended environments that are

sub sets of any one or more of the following: a respective Enterprise Information System (EIS) application, a client environment, and one or more intermediate server environments.

In another preferred embodiment, the package structure 1190, e.g., one with any of the asset combinations described herein, include assets that are parts (or all) of one or more base environments. Some of these assets include any one or more of the following: a web server for static content assets (SC), a servlet engine for JSPs (JSP), a Java Runtime Environment for Java class assets (JB), an application server for the EJB assets (SB and EB), and a DBMS for the data assets (RD, ED, and EB).

In a preferred embodiment, a minimum application server (MAS) is used in the base environment. The MAS provides the minimal services that an application needs when distributed to a client. This may includes naming, web, servlet, transactional, and database services. These services are minimal because they may not have proprietary or extended extensions that the services might have in the EIS environment.

In another preferred embodiment, the package structure 1190, has assets that comprise one or more of the following agents: a publishing agent, a subscriber adapter, a caching agent, and a computational agent.

In another preferred embodiment, the package structure 1190, has one or more assets that comprise any one or more of the following adapters: a discovery adapter, a versioning adapter, an export adapter, a process adapter, a target adapter, a client deployment adapter, a synchronization adapter, a bridging adapter, an adjustment adapter, a streaming adapter, a quality of service (QoS) adapter, and an asset packaging process.

In another preferred embodiment, the package structure 1190 has one or more of the assets of a reference data and/or entity data asset type with a logic/data part that is data based on one or more queries and with an extended environment that is table schema.

In one preferred embodiment the package structure 1190 has one or more transactional components asset types. These transactional component asset types can include:

a. one or more assets having an asset adapter based on a CDS/ADS adapter asset type, with a logic/data part that is one or more asset adapter class files that each support one of the respective asset types, and with an extended environment part that contains licensing information.

b. one or more assets having a java server page (JSP) asset type, with a logic/data part that is a JSP and with an extended environment that is one or more JNDI entries.

c. one or more assets having a java class file asset type, with a logic/data part that is java class file and an extended environment that is one or more JNDI entries.

d. one or more assets having a session bean asset type with a logic/data part that is an enterprise java bean (EJB) implementation and with an extended environment that is an EJB stub and skeleton deployment descriptor and JNDI entries.

e. one or more assets having a java entity bean asset type with a logic/data part that is an EJB implementation based on a query and the extended environment that is an EJB stub and skeleton deployment descriptor and JNDI entries.

f. one or more assets having a Smalltalk asset type with a logic/data part that contains a Smalltalk image.

g. one or more assets having a C++ asset type with a logic/data part that is an executable file and with an extended environment that is one or more registry entries or environment variables.

h. one or more assets having a C++ asset type with a logic/data part that is a dynamic link library (DLL) and with an extended environment that is one or more registry entries or environment variables.

i. one or more assets having a Perl asset type with a logic/data part that is Perl script and with an extended environment that is one or more environment variables.

In one preferred embodiment the package structure 1190 has one or more static content asset types. These static content asset types can include where:

a. the static content asset type is an HTML page.

b. the static content asset type is an HTML page with an extended environment part that includes a web server alias.

c. the static content asset type is any one or more of the following: a JPEG file, a GIF file, Java Applet, a SVG (Scalable Vector Graphics), a PDF (Portable Document Format), a TIFF (Tag Image File Format), an EPS (Encapsulated Postscript) file, a PNG (Portable Network Graphics) file, an XML file, a WML (Wireless Meta Language) file, a BMP (Bit Map) file, an XHTML (extended HTML) file, a DHTML (Dynamic HTML) file, an MPEG (Motion Picture Expert Group) file, an AVI file, and any static content transferable via an HTTP protocol.

d. the static content asset type has an extended environment that contains a web server alias.

e. one or more of the assets in the package has a music asset type with a logic/data part that is an MP3 file and with an extended environment that is one or more sets of album and track information.

f. one or more of the assets in the package has a video asset type with a logic/data part that is an MPEG file and with an extended environment that is one or more production notes.

g. one of the assets in the package has a document asset type with a logic/data part that is a PDF file and with an extended environment that is one or more authentication certificates.

In an alternative embodiment, the package structure can contain one or more assets having a relational data asset type, e.g., a reference data (RD) and/or an entity data (ED) asset type and one or more other assets that have a presentation component asset type. In particular, these presentation component asset types can be any one or more of the following types: a dynamic content (DC), a java bean (JB), a session bean (SB), and an entity bean (EB). In addition, the package structure further can comprise one or more still other assets that have a transaction component asset type. In a preferred embodiment, these transaction component asset types can be selected from any one or more of the following types: an entity bean (EB), a java bean (JB), a session bean (SB), and a dynamic content (DC).

In one preferred embodiment, the package structure has one or more of the assets that is an asset adapter based on a CDS/ADS adapter asset type that has a logic/data part with one or more asset adapter class files that each support one of the asset types, and with an extended environment part that contains versioning information.

Figure 4 is a block diagram of one preferred asset definition data structure 1170. In this embodiment, each of the records, typically 1175, of the asset definition data structure 1170 contain an asset identification field 1172. The asset identification field 1172 uniquely identifies the asset for a given system. In cases where there are multiple systems, a system identifier (or some other designation) might have to be included in the asset identification to uniquely identify the asset. Any unique identifier can be used in the asset identification field 1172. Another field contains information that further identifies the particular asset with an asset name 1176.

Other fields can give more detailed information about the asset. For example, the argument string field 1174 gives information that can be used to determine where to obtain the asset, e.g. the URL/URI of the asset. Machine identification information and/or other location and/or memory locations could also be used in field 1174. An optional name field 1176 can provide more textual description of the asset, similar to the asset ID 1172. A type field 1178 gives specific type (asset type 240T) information about the asset, e.g. static content assets (SC), session beans (SB), entity beans (EB), java server pages (JSP), java beans (JB), reference data (RD) and entity data (ED). Asset versioning information 1012V, e.g. a time stamp for the asset 1012, can be provided in an asset versioning field 1179. Other information optionally can be included 1179A.

Figure 5 is a block diagram of a deployable asset data structure 1370. Each deployable asset data structure record 1375 associates an asset 240 (also see Figure 10, 1012) with a current version of the respective asset. In a preferred embodiment, this is done by associating an asset ID field 1372 and a version field 1374 in the asset data structure record 1375. Typically, the asset ID field 1372 contains the asset ID 1172 of the asset definition data structure (1170,

1172) (and the package content data structure 1190, 1194). The version field 1374 contains the versioning information 1179 in the asset definition data structure 1170.

Figure 6 is a block diagram of a client/target asset table 1390. The client asset table 1390 associates the client nodes/targets 940 (see Figure 9) with a set of client assets for the respective client node 940. In a preferred embodiment, each client asset table record 1395 of the client asset table 1390 contains a client ID field 1392 and a client asset field 1394. Typically, the client ID field 1392 contains the node identification (node ID) information 1382 for the respective node 940, or may be used to indirectly look up the client node ID in another data structure. The client asset field 1394 contains the asset ID 1172 of the asset or assets that are to be associated with the respective node 940, this asset ID may be used to look up the actual asset information in another data structure. For example, if the client node 940 were to subscribe to a particular quality of service (QoS) or particular program, e.g. a movie, the assets (240, 1012) needed to provide the client node 940 with this QoS or program would be defined in the client asset table 1390. Thus, in this example, the client/target asset table 1390 is a list of the assets 240 (and packages) to which the target has subscribed.

(Note that in this disclosure, the term "client" is often used to means "target" without loss of generality. A client or target is one or more of the machines on the network where the asset 240 is to be delivered at any point in the life cycle of the asset. Therefore, a target can include: one or more clients, proxy servers, applications servers, CDS/ADS servers, EIS's, or computers within EIS, or even applications within the computer or computer system on which the asset currently resides. Generally, a target/client is any computer (or node) that is capable of communicating over the network.

Figure 7 is a block diagram of a client/target deployment queue 1380. In a preferred embodiment, the client/target deployment queue 1380 is a list of nodes/targets or client nodes 940, e.g. clients, servers, proxy servers, localized servers, slave servers, forefront servers, source EIS systems, and/or target EIS systems, to which packages/assets are to be deployed. In a preferred embodiment, the client/target deployment queue 1380 is located on the CDS/ADS. Each record 1385 of the target/client deployment queue 1380 typically contains the target/node identifier (node ID) 1382. The node ID can be any designation that uniquely identifies the node including: a node name, a node machine address, node Lightweight Directory Access Protocol (LDAP), node network name, or any other identifier that uniquely identifies the respective node 940. In an alternative embodiment, the node ID 1382 has one or additional fields, or, is an XML file that includes the above information in addition to other information about the node/client/target. This additional information will identify the node/client/target machine operating system. Other optional information can be included.

Figure 8 is a block diagram of one preferred physical architecture 800 of a network system that uses the present invention. The system 800 comprises one or more networks 850, in a preferred embodiment, the Internet (or other network) 850 that is connected to one or more EIS tiers 810, one or more component distribution servers/asset distribution server (CDS/ADS) tiers 860, and one or more target/client tiers 830 (of one or more clients, typically 835) using any well known configuration and communication connections. Each of the EIS

tiers 810, CDS/ADS tiers 860, or the client tiers 830 can be further divided into sub-tiers as is done in many well-known network designs. The CDS/ADS tier 860, or portions of its functions, may reside on the EIS tier 810. There can be a plurality of EIS 810 functions that may or may not be connected through the network 850.

In one preferred embodiment, the EIS tier 810 can comprise one or more sub-tiers, e.g., a web tier (e.g. a web server) 802, a transactional tier (e.g. a transactional server) 804, and a database tier (e.g. a database server) 806. All of these functions could be located on one server or broken up and distributed across any number of servers connected through the network 850 or by any other known connections. The web tier 802 produces the web pages that are served to the web client(s) over the network 850. The transactional tier 804 typically runs application programs that are EIS and/or client specific and whose results and/or input are communicated to/from the target/client(s) 835 over the network 850. The database tier 806 contains information that is used by the transactional tier 804 and that may be (partially) accessed directly by the client tier 830. The database tier 806 includes EIS data like legacy data and may also include data that is available from any other storage device or server connected to the network – whether part of the EIS 810 or not.

The EIS tier 810 can be any computer system that is used in any business. Generally, the EIS can store and process information and provide information as outputs or control functions. The EIS tier 810 can be used in any general product or service business, manufacturing facility, or other general organization. These businesses and organizations can include both traditional business and web/internet based businesses and range from businesses and business functions dealing with raw material acquisition and handling, research and development, product manufacturing, product distribution and storage, marketing, retail and wholes sales, customer relations, advertising, accounting, finance, taxes, business-to-business transactions, media, maintenance, equipment control, inventory, and any other product and service related to the business. The EIS tier 810 can be used in any general business type providing products and/or services including: government, legal, engineering, financial, mortgages and loans, banking, stock and security brokerages, education, real estate, financial, food, and manufacturing equipment control.

The CDS/ADS server 860 can perform standard middleware functions but also provides the novel facilities for breaking up and recomposing programs or program fragments into assets 240 (or sub assets) and distributing these recomposed assets 240 (sub assets) to any tier, platform, and/or environment for execution. See below description.

The targets (e.g. clients, etc.) are in a target tier and can be any known target 835 that is served or sent information over the network 850. In a preferred embodiment, the target/client 835 has a client agent and one or more particular client architectures that novelly allow for the remote execution of portions of the EIS business logic on the respective client 835. The target tier 835 can include any general client computer 830 including a personal computer at the site and/or residence of a client/customer, a point of sale computer for providing customer information, a kiosk, a local (e.g. a LAN) server, a proxy server, a controller on a piece of equipment, or even another EIS tier 810 or server on the network 850.

Figure 9 is a block diagram of one preferred logical architecture 900 of a network system using the present invention. (In this disclosure, this architecture 900 is also referred to as the Distributed Internet Services or DIS 900).

The EIS tier 810 has an EIS tier architecture 910 that includes one or more EIS servers 915. One or more of these EIS servers 915 has various configurations of EIS agents and/or adaptors 920 that perform functions described below.

As described below, the EIS adapter/agents 920 are able to process any general application parts and/or asset 240. (In some embodiments, the EIS adapter/agents are part of the Component Distribution Server (CDS/ADS) tier and are distributed to the respective EIS to perform their functions.) Again, the EIS tier architecture 910 can comprise one or more sub-tiers and/or one or more different EIS's 910. The EIS tier 910 communicates with the client node tier architecture 935 on the client node tier 830 over network connections 992 by any appropriate protocol 992P. In a preferred embodiment, these protocols 992P include the well-known web protocols, web page protocols, browser/client protocols, network communication protocols, and connection protocols.

In a preferred embodiment, the EIS 910 communicates over network connections 994 with one or more CDS/ADS tier architectures 960 in the CDS/ADS tier 860. The communication between the EIS 910 and the CDS/ADS tier architecture 960 includes any appropriate protocol 994P including CORBA – Common Object Request Broker Architecture; IIOP - Interoperable Internet Object Protocol; RMI – Remote Method Invocation (T3, JRMP – Java Remote Interface Protocol, IIOP). The CDS/ADS tier architecture 960 also communicates with the client node tier architecture 935 through network communication connections 996 using protocols 996P. These protocols include CORBA (IIOP), RMI (T3, JRMP, IIOP), and Multi-platform Communication Connection.

Note that technically, the Component Distribution Server (CDS) typically processes component 120F level system part 100F or larger. Alternatively, the Asset Distribution Server (ADS) processes digital assets 240 of any type in the system part 100F. However, in this disclosure, the terms CDS, ADS, and CDS/ADS may be used to describe either of these two embodiments, without loss of generality.

The asset Server Tier architecture comprises one or more asset server nodes 955 that may be distributed across two or more sub-tiers. One or more of these asset server nodes 955 has a package specification process 1200 that could involve various combinations of CDS/ADS agents 964 and adapters 966 (as described below). Generally, in one preferred embodiment, the CDS/ADS agents/adapters 964/966 find assets in respective tiers across the network and package these assets for distribution to the clients 835 on the client tier 830 and/or across other tiers of the network. See description below.

In one preferred embodiment, the asset server node 955 has a collaboration function 970, often provided by a collaboration server 970. The collaboration function can be any known

collaboration function in the art. This could include U.S. Patent 6,240,444, entitled "Internet Web Page Sharing" to Fin et al. issued on May 29, 2001 or U.S. Patent application number xxx, entitled "System and Method for Collaboration Using Web Browsers" to Chen et al., filed on xxx, that are both herein incorporated by reference in their entirety. Note that the collaboration function can be performed by handling the TDS like any other asset 240 that is packaged, distributed, executed, synchronized, and/or managed through an asset life cycle 240L.

As stated above, the target tier 935 architecture includes one or more client nodes 940. The target/client node tier architecture 935 can include many different types of clients 940 including: personal computers, workstations, pervasive devices, local servers (e.g. LAN servers), proxy servers, general network servers, EIS systems, etc. Clients 940 in the target tier 935 include any generally known client architecture. The target tier 935 and client 940 architectures can be distributed throughout the network, can be divided into sub-tiers, and can even include general network servers (like 802, 804, 806) or EIS systems 810 functioning as a client 940 for a particular application, sub application, component, module or function. These client nodes may comprise one or more novel client distribution agents or adapters (CDA) 944, in various configurations (see below), that handle the communication between the client node and either the CDS/ADS tier 960, other client/target tiers 935, and/or one or more EIS tier 910.

Figure 10 is a block diagram of an example data structure 1000 showing a plurality of packages1012. In most cases, one or more packages, typically 1050A, make up an application program 1050 or a portion of an application program 1050. The packages are made of one or more assets 240.

The data structure 1000 shows one or more packages, P1 ... Pp, 1050A. Each of the packages, typically 1050A, comprises one or more assets, typically 1012 for example package 1050E (e.g. example assets 1002, 1004, 1006, 1008, 1009, 1010, 1011, 1014. 1016, 1018, and 1020). Typically, the assets (240, 1012) in a package 1050A have an expressed or implied logical relationship, i.e. a call graph. In the example package 1050E, a.jsp 1002 can call a navigate.jsp 1004. The navigate.jsp 1004 can in turn call a "chuck.class" 1006 object via edge 1032, a shuang.class 1011 object via edge 1034, or any Java Bean (a Sun Microsystems Java language construct) 1014 via edge 1036. The "chuck.class object 1006 can call an entity bean 1008 that can access entity data 1009 in a relational database. The Java Bean 1014 can call a session bean 1016 which can call an entity bean 1018 which can access reference data in a relation database as asset 1020. In some preferred embodiments, the call graph information might be included as additional information in the package (below). In other embodiments, the call graph can be implied.

In the context of the CDS/ADS, the term package 1050A refers to a logical collection of assets 240. These assets can be grouped following different criteria, such as locality of execution or the generation of some output. Within the context of a preferred embodiment of a web application, a package is a grouping of the assets that are needed to generate the output

for one or more web pages. It is convenient to refer to these packages based on the URL associated with the request for the web page the assets generate.

The structure above might be used through a web page. The web page would allow a user to see the balance of various accounts, such as checking, savings, and retirement accounts. The page would need to access the JSP pages to build the user interface. Those JSP's would access the class files in order to perform some intermediate functionality, such as sorting account entries in the summary. The session bean 1016 would be required for managing the data associated with the user's connection, and possibly accessing data in the relational database, such as the account balances. The entity bean 1018 would store information that represents a single data entity, such as a stock price table that provides the current value of stock prices used to calculate the balances of the retirement account.

Note that any one of the assets (240, 1012) in this package (1050E, 1050A) can have a version 1012V. The version 1012V can be any known way of distinguishing the asset 1012. In a preferred embodiment, the version is a time stamp. Other examples of versions 1012V include locations, machine/node number, source, destination, checksum, differencing algorithm, file name, file system extended attributes, other versioning information, etc., or combination of these.

While packages 1050A can include the actual assets 1012 of the package 1050A, in a preferred embodiment, this is not done. Rather, some identifier of the asset 1012 is included in a list. See the description of one preferred embodiment in Figure 3 and Figure 3A above.

Figure 11 is a block diagram of an example data structure 1100 for package definition. The data structure has a plurality of package specification records, typically 1105. Each package specification record 1105 has a package identification field (package ID) 1110 that contains information used to identify the package of the particular record 1105. This package identification (package ID) 1110 field typically is a package name, but could be any other way to identify the package, e.g. a number of the package location, object identifier, URL/URI name, etc. In a preferred embodiment, the package ID is the URL/URI name associated with the package (or asset – see below) on the EIS.

In addition, the package definition data structure 1100 optionally includes timing information 1150. The timing information 1150 includes any information that determines when the respective package is delivered to one or more locations over the network. There are many (optional) ways to designate this timing information 1150. For example, an "immediate" designation 1152 specifies that the package should be delivered over the network as soon as the package is available for delivery (i.e. when the package is specified). In an alternative embodiment, a delivery start time 1154 can be provided in the package definition data structure 1100 to provide a time for the package to be delivered over the network. In this alternative embodiment, the "immediate" designation field 1152 might be omitted or alternatively, provided with a value equal to "not immediate."

In another embodiment, if no package timing 1150 information is given, the package can be immediately sent out.

In another embodiment, a delivery end time 1156 of the package is given. In this embodiment, the package is scheduled to be sent over the network at any time prior to the delivery end time 1156.

Other variations are possible. An expire time 1158 optionally can be provided to indicate a time beyond which the particular package 1105 should not be sent. A remove time 1160 can be provided to indicate a time beyond which the record 1105 should be deleted, or marked unusable in the package definition data structure 1100.

In another embodiment, a refresh rate can be provided designating how often the record 1105, or table 1100, should be updated.

Variations of one or more of these fields of information 1152, 1154, 1156, 1158, 1160, and/or 1162 can be used as the timing information 1150.

Information about the location of a particular package can also be used. In a preferred embodiment, the package location information 1120 is stored in the package definition data structure 1100 for each of the package records 1105. Typically, this location information is the location at which the package can be found on the network. In a preferred embodiment, the location information is the uniform resource locator (URL/URI) of the package. Just the network host name can also be used to represent the location of the package as well.

In still other embodiments of this invention, the package definition data structure 1100 does not need to define packages of multiple assets at all. In this embodiment, the data structure 1100 defines one or more single assets (as a package) 1012, e.g. where the package id 1110 is replaced by an asset id (1172 below.) Also, this can be thought of as one or more of the package records 1105 referring to a package of only one asset 1012. In some of these cases, the package definition data structure is really a data structure that defines assets, e.g. one record 1105 per asset. However, in more preferred embodiments, one or more of the packages contain more than one asset. In preferred versions of these embodiments, the timing information and information defining the assets (e.g. asset type 1178, below) is required.

In this document, the description of data structure 1100, whether used for assets and/or packages (of assets), will be referred to as package definition data structure that define assets and/or packages without loss of generality.

In a preferred embodiment, the package content data structure is specified as an XML document. There is an XML DTD (Document Type Definitions) associated with the document that specifies the allowed structure.

A novel Application Markup Language (AML) refers to any XML document that adheres to the XML DTD defined for the package content data structure specification. HTML is defined

in a similar manner and supports the specification of resources (e.g. text, other HTML documents, images) that are to be gathered and displayed within the document. AML can be used as a greatly expanded HTML. Since HTML covers only a small number of resources, the capability to define plug-in modules for both the browser and the web server have allowed HTML to handle a greater range of applications.

Utilizing AML, the DIS is able to support many more capabilities than the plug-ins allow (e.g. servlet, EJB, RDBMS). In some embodiments, the AML specifies the assets (resources) an application needs to display a single page in a browser. The distribution of these assets to the client environment is analogous to the download of assets using the HTTP protocol. However, using AML, any general web and/or enterprise application (especially those defined by the J2EE specification) can be distributed from the server to the client and be executed in the client environment with out the need for additional plug ins. Further, using the infrastructure disclosed in this disclosure, all supporting environment is provided for the deployment of any of these general assets on any given client without the need for specialized adaptors/plug-ins. In addition, using this novel infrastruture, designed from the ground up to support web and/or enterprise applications, any general enterprise application can be discovered, processed, and distributed over any tier of the network so that these enterprise applications can be deployed on any given client on the network and the coherency of the applications and/or data at the enterprise system remains in tact.

Figure 11A is a block diagram of alternative preferred package data structure 1100A with an extended environment for packages (EEP). This data structure has an extended environment package (EEP) 1120A and one or more assets 240, or more preferably, asset identifiers, e.g., 1172. In a preferred embodiment, the EEP 1120A is an XML file as described above for the digital asset EE 220 alternative. There will also be descriptors in the EEP 1120A. However, these descriptors will normally apply to features that all digital assets 240 identified in the package data structure 1100A have in common.

For example, here is an possible EEP 1120A:

```
<extended_environment package>
        <common> 1110A
                <name></name>
                <address></address>
                <size></size>
                <volatile></volatile>
                <runnable></runnable>
                <version></version>
                <user_type></user_type>
                <security></security>
                <priced></priced>
        </common>
        <dependencies> 1121A
                <package_dependencies> 1122A
```

```
                    <package></package>
                </package_dependencies>

            </dependencies>
            <references> 1160A
                <reference_links></reference_links>
                <reference_files></reference_files>
                <reference_directories></reference_directories>
            </references>
            <pricing> 1175A
                <actual_price></actual_price>
                <price_scheme></price_scheme>
                <payment_details>
                    <payment_method></payment_method>
                    <credit_card_info></credit_card_info>
                </payment_details>
            <pricing>
            <security> 1180A
            </security>
</extended_environment package>
```

In an alternative preferred embodiment, the EEP 1120A may have one or more common descriptors 1110A. These common descriptors could include a package name, address, and/or size. The common descriptors could give information common to all the digital assets 240 in the package 1100A. For example, are all of the digital assets 240 in the package volatile, runnable, the same user type, and/or the same version. Common descriptors could also indicate package level security or pricing information.

The EEP dependencies descriptors 1121A can include package dependencies 1122A that describe other packages with which the package data structure 1100A is associated.

In an alternative preferred embodiment, the EEP 1120A also has one or more reference descriptors 1160A. In a preferred embodiment, the reference descriptors 1160A include any one or more of the following at a package level: a reference link descriptor, a reference file descriptor, and a reference directory descriptor. The reference link descriptor provides a world-wide-web (WWW) address that has contents used for processing of the package. The reference link descriptor provides a world-wide-web (WWW) address that has contents used during execution of all the digital assets in the package. The reference file descriptor is a unique fully qualified name of a file required for reference by the package. The reference directory descriptor provides additional address information that is used to locate all the associated digital assets in the package. For example, the additional address information could include root or parent level directory names, which would, following well known format, be combined with the leaf directory level identification given in the common name descriptor to locate and fully qualify the path information for the digital asset.

In an alternative embodiment, the EEP 1120A further comprises or more pricing descriptors 1175A. The pricing descriptors describe information about any well-known general pricing information at a package level including one or more of the following: a price, a price scheme (subscription, pay to own, pay to use, one time payment), a payment detail, and payment method (check, credit card, card number).

In an alternative preferred embodiment, the EEP 1120A further comprises one or more security descriptors 1180A. The security descriptors are well known and may describe any one or more of the following functions at a package level: encryption, authorization, and access control.

Other EEP 1120 descriptors are envisioned.

Figure 11B is a UML diagram 1100B showing the transitive part-whole associations between certain digital assets and certain packages. In this figure, a StaticPackage class 1105B that must at least one StaticAssets 282C as a member, and can contain assets of no other type. A RunnablePackage class 1110B must have at least one RunnableAsset 284C as a member but optionally can have StaticAssets 282C and VolatileAsset 280C. A VolatilePackage 1115B must have at least one VolatileAsset 280C and optional can contain StaticAsset 281C.

Figure 12 is a flow chart of a package specification process 1200. The package specification process 1200 determines whether a given package is ready for immediate delivery, in which case the assets in the package are packaged and delivered, or if the package needs to be scheduled for later delivery. In a preferred embodiment, the delivery/schedule process 1200 uses a message queue to "decouple" the schedule and delivery of one or more of the packages from other functions of the system.

Steps 1205, 1210, and 1215 of the delivery/schedule process 1200, are an optional set of steps that uses the message queue of the system to place package specification records 1105 into the system database. These steps permit the system to do other functions without regard for the time of receiving or storing the package specification records in the database – i.e., the decoupling. First, the package specification record 1105 is written 1205 to the message queue. As the system processes the message queue, the package specification record 1105 is read 1210 from the message queue and then written 1215 to a package specification database record (in the system database) of the same format as the package specification data structure 1100.

In a preferred embodiment, the package specification record 1105 is written to the message queue by some automated process, or, using a tool to aid in the specification. See description of the export process below.

Step 1220 examines the copy of the package specification records 1105 in the system database to determine if the package specified by the package specification record 1105 is ready for immediate deliver. This can be done in various ways. In one embodiment, the immediate

field 1152 is examined. If the immediate field 1152 has a value of "immediate", the package specified by the package specification record 1105 is sent 1230 to the asset packaging process 1300 (see Figure 13). If not, the specified package is scheduled 1235. In a preferred embodiment, the specified package is scheduled by identifying the package in a packaging queue. See data structure 1250 describe in Figure 12A.

In alternative embodiments, delivery readiness (1220, 1225) is determined by comparing the current time to various fields in the timing information 1150 of the package definition data structure 1100. For example, if the current time is greater than the delivery start time 1154 the package is sent to the asset packaging process1230- if not the package is scheduled 1235. Alternatively, if the current time is greater than the delivery start time 1154 and less than the delivery end time 1156, the package is sent to the asset packaging process1230- if not the package is scheduled 1235. Or, if the current time is less than the delivery end time 1156, the package is sent to the asset packaging process1230- if not the package is scheduled 1235 or deleted. Other "delivery immediate" checks 1225 are envisioned. In a preferred embodiment, the package is scheduled by placing the package in a packaging queue 1235. In an alternative embodiment, the packaging process can be done before the package timing information is even examined, leaving that examination for deployment time.

Figure 12A is a block diagram of an example preferred packaging queue 1250. The packaging queue 1250 has one or more packaging queue records 1255. Each of these packaging queue records 1255 has a package queue package ID 1252 and a package queue start time 1254. In the preferred embodiment, step 1235 places the information in the package ID field 1110 into the package queue package ID 1252. In a preferred embodiment, the queue start time 1254 is determined by a deployment engineer. The packaging queue is used to hold requests for the deployment of packages until they are scheduled to be deployed.

In a preferred embodiment, the data structures described in Figure 3, 4, 5, 6, 7, 11, 11A, 11B, and 12A are embodied as tables or views of any well known relational database management systems, e.g., IBM's DB2, Oracle, Microsoft's SQL Server, Sybase, Informix, etc. This permits operations on the data structures using known database techniques, for example SQL queries in any known relational database manager.

Figure 13 is a flow chart of an asset packaging process 1300. The asset packaging process 1300 packages the assets that make up the packages that are to be delivered immediately 1230 or that are scheduled for later delivery 1235 if the start time 1254 is less than or equal to the current time, or if any other delivery requirements are met. The asset packaging process takes as input the records 1255 from the packaging queue 1250 (in the case of the "later" delivered packages 1235) or the package ID 1110 of the record 1105 that is being delivered immediately 1230.

The asset packaging process 1300 first identifies 1305 the pending packages (or assets, if a "single asset package"). A pending package is any package on the packaging queue 1250 whose start time is less than or equal to the current time, any package that is known to need immediate delivery, or any package meeting any other delivery requirement.

Optional step 1310 determines which of the pending assets needs to be delivered at the current time. A pending asset is any asset in a pending package. In a preferred embodiment, the pending asset is any asset in a pending package that is not already located at any node/client 940 requiring the respective asset.

If there are no pending assets, the process 1300 waits for a period of time and then performs the check 1310 again. This period of time is determined by application requirements. In a preferred embodiment, the period of time is the time between the current time and the time of delivery for the asset closest in time for scheduled delivery, e.g. listed in the packaging queue 1250.

If it is determined that there are pending assets, the process proceeds to step 1315 which makes the process 1300 consistent while the assets are being packaged for delivery. In one embodiment, the client/target deployment queue 1380 is locked to achieve consistency. In an alternative embodiment, a distributed lock is used to support clustering of the CDS/ADS. In other embodiments, any standard known contention resolution method is used, e.g. transactional locks, synchronization, semaphores, mutual exclusion, etc.

In an alternative embodiment, if there are pending packages/assets that need to be distributed, any changes in the nodes (e.g. clients) 940 that are to receive these packages/assets 1012 have to be fixed for a while in order to maintain consistency. In a preferred embodiment, the "lock" 1315 on a client/target deployment queue 1380 accomplishes this. The "lock" 1315 prevents any writing to or deleting from the client/target deployment queue 1380 until the lock is removed. Locking data structures for consistency is well known.

In step 1320, the pending assets are determined. The assets are determined for the pending packages. In a preferred embodiment, only those assets that are in a pending package that are not included in one or more nodes/client register for the respective asset are determined as being pending assets. In a more preferred embodiment, only those assets that are in a pending package that are not included in one or more nodes/clients 940 registered 1390 for the respective asset (1012, 1394), and are not in an asset cache, e.g. accessible by the server, are determined as pending.

In a preferred embodiment, this determination is done by logical intersections between the set of assets in pending packages and a set of registered assets (associated with specific nodes). This can be done by using well know database techniques, for example SQL queries in any relational database manager. See below.

In one embodiment, some of the information in the package definition data structure 1100 is specified by an external system in the form of a text based data structure such as XML. Additionally, the information contained in the package record 1105, the package content data structure 1190, and the asset definition data structure 1170 is represented in relational database tables for the purpose of performing set operations on this and other data in the CDS/ADS data store (the database). For example:

The reference numbers in the following SQL create statement correspond to the reference numbers in Figure 11:

```
CREATE TABLE PACKAGEDESCRIPTOR ( 1100
        PACKAGE_ID          VARCHAR( 32 )   NOT NULL, 1110
        IMMEDIATE           CHAR( 1 )       NOT NULL, 1152
        URL                 VARCHAR( 200 )  NOT NULL, 1120
        REFRESH_RATE        INTEGER, 1162
        START_DT            TIMESTAMP       NOT NULL, 1154
        STOP_DT             TIMESTAMP, 1156
        EXPIRED_DT          TIMESTAMP, 1158
        REMOVE_DT           TIMESTAMP, 1160
        PRIMARY KEY (PACKAGE_ID)
CREATE TABLE PACKAGEASSETS ( 1190
        PACKAGE_ID          VARCHAR( 32 )   NOT NULL, 1192
        ASSET_ID            BIGINT          NOT NULL, 1194
        PRIMARY KEY (PACKAGE_ID,ASSET_ID)

CREATE TABLE ASSETDESCRIPTOR ( 1170
        ASSET_ID            BIGINT          NOT NULL, 1172
        ASSET_TYPE          VARCHAR( 3 )    NOT NULL, 1178
        ASSET_NAME          VARCHAR( 100 )  NOT NULL, 1176
        TIMESTAMP           TIMESTAMP       , 1179
        PRIMARY KEY (ASSET_ID)
```

The use of primary keys as in the above tables is a well known way of defining how a table is stored and accessed.

For the purpose of this embodiment, two relational database views, VIEW_S1 and VIEW_S2, are created. Step 1320 uses these views to determine which assets are candidates for client deployment. These views are described as follows:

```
CREATE VIEW VIEW_S1 (PACKAGE_ID, URL) AS
        SELECT PACKAGE_ID, URL FROM PACKAGEDESCRIPTOR
        WHERE START_DT < CURRENT TIMESTAMP

CREATE VIEW VIEW_S2 (ASSET_ID) AS
        SELECT DISTINCT ASSET_ID FROM PACKAGEASSETS
        WHERE PACKAGE_ID IN
        (SELECT PACKAGE_ID FROM VIEW_S1)
```

VIEW_S1 is a subset of the data contained in the PACKAGEDESCRIPTOR table (e.g. 1100). The subset is obtained by selecting those package id's that correspond with start dates later than the current date.

VIEW_S2 is a subset of the data contained in the PACKAGEASSETS table (e.g. 1190). The subset is obtained by selecting those asset id's in the PACKAGEASSETS table that have package id's contained in view VIEW_S1.

Optional step 1325 clears a deployable asset data structure 1390 (in the example: ASSETDESCRIPTOR table). This is done to prevent duplicates in the table and to prevent the table 1370 from over filling.

In an alternative embodiment, only the entries in the deployable asset data structure 1390 associated with the package or packages being distributed are cleared. This is done to prevent duplicates in the table and to prevent the table 1370 from over filling.

Step 1330 gets the list of pending assets (result of intersection, selecting the contents of the VIEW_S2 view into the ASSETDESCRIPTOR table 1170). Step 1335 sends a request to the EISs for the current version of the assets on the list. See description below. In application where the version of the asset is unimportant, step 1335 can be omitted. The version/timestamp field 1179 of this table 1170 is left empty.

In an alternate embodiment, the SQL queries used to define the views could be combined into a single SQL query. Such a query could be executed to obtain the results and insert those results into the final ASSETDESCRIPTOR table.

In a preferred embodiment, step 1335 sends the list of pending assets to the EIS distribution agent. The EIS distribution agent then updates 1340 the list with the current version. In one embodiment, the EIS distribution agent compares the asset Ids (e.g. file names) on the list to the current versions in the EIS. The EIS distribution agent then writes the current version of the respective asset (asset ID) to the list 1170 in the version field 1179. The list is then returned 1340 to the CDS/ADS by the EIS export agent (see the description of Figure 16, 1600, below).

The CDS/ADS 965 uses the list 1170 with the current version of the assets 1179 to update the deployable asset table 1370. In one preferred embodiment, the CDS/ADS then determines the "final" deployable assets, which are the current versions of the assets in the deployable asset table minus the assets in the CDS/ADS asset cache that match. This is done by subtracting those assets in the deployable asset data structure 1370 that are available to the CDS/ADS 965 in a cache memory. This final deployable asset list 1370 is sent to the appropriate EIS 810. For each asset (1372, 1012) on the list 1370 an export asset adaptor (1600 below) is called for that asset, based on the asset type. The asset adaptor persists the respective asset to a data structure that the EIS distribution agent 1600 sends to the CDS/ADS. The data structure is determined by the asset type specific to the asset that is being used.

In step 1345, the CDS/ADS 965 makes a request of the EIS export adaptor/agent (1600 below) to provide the assets 1012 listed on the current deployable asset data structure 1370. Once these assets 1012 are received from the respective EIS 810, the CDS/ADS 965 stores 1350 these assets 1012 in one or more memories available to the CDS/ADS 965.

In a preferred embodiment, a cache is used to store the assets on disk or in memory. The CDS/ADS 965 then writes the deployable asset data structure 1370 to the asset cache data structure in the CDS/ADS. The CDS/ADS 965 writes 1355 the client Ids 1382 that require assets 1012 to the client/target deployment queue 1380.

In a preferred embodiment, the CDS/ADS 965 now has cached in one or more available memories all of the assets 1012 required by all of the client nodes 940 at the current time.

At this point, any locking step is reversed 1360, e.g. the client/target deployment queue 1380 is unlocked.

In other embodiments, the client locks are also removed 1360 at this time.

In the context of the CDS/ADS, the term agent refers to a process or set of processes executing on a node, responding as a server and tasked with completing some significant job on behalf of the requesting process.

Figure 14 is a flow chart of a client deployment process (CDP) 1400. The client deployment process 1400 is used to deploy assets from the CDS/ADS 965, or distribution tier 960, to a respective client node 835. These assets 1012, or asset fragments (as described below) 1012, are moved from the CDS/ADS 965 to the client node 835 as data structures, and the movement is achieved through well known network programming methods. In a preferred embodiment, a client distribution agent (CDA) 944, e.g. deploy, runs on a respective client node 835/935 in order to determine whether assets need to be obtained from the CDS/ADS 965.

The client deployment process first 1400 determines if there are any assets 1012 pending for delivery to a specific client node 835. In a preferred embodiment, the client deployment process 1400 queries 1420 the client/target deployment queue 1380 for its client identifier. If the identifier is on the queue, this indicates that the CDS/ADS has assets pending for delivery to the respective client node 835/935. Here, a pending asset is any asset that has been identified and moved from the EIS 810 to the CDS/ADS 965 in steps 1320-1355. And in a preferred embodiment, the asset (240, 240P, 1012) is associated with a package that has a delivery timestamp equal to or less than the current time, and which has not already been deployed to the client. The process is able to determine if the asset has been delivered to the client by checking the list of assets and their timestamps, which is stored in the client asset database table 1490 described in figure 14B and which resides on the server.

Figure 14B is represented by a client asset table resident on the CDS/ADS and, in a preferred embodiment, as a database table structure as defined in the table create statement, below.

```
CREATE TABLE CLIENTASSETS (
      CLIENT_ID          VARCHAR( 32 )   NOT NULL,  1452
      ASSET_ID           BIGINT          NOT NULL,  1454
      TIMESTAMP          TIMESTAMP       NOT NULL,  1479
PRIMARY KEY (CLIENT_ID, ASSET_ID)
```

The client deployment process 1400 contacts the CDS/ADS 965 in step 1410 in order to determine if there are assets (240, 240P, 1012) pending for the client node in step 1420. Each client node (target) 940 has a client id 1382 associated with that node (835, 940). The client node (835, 940) asks 1420 the CDS/ADS 965 if there is a message corresponding to the client id 1382 in the client/target deployment queue 1380.

In step 1430 the client node agent residing on the client 1400 determines if a message was pending in the client/target deployment queue 1380. If a message is not pending, the client deployment process 1400 proceeds to step 1440. If a message is found (is indicated) in the

client/target deployment queue 1380 for the respective client node (835, 940), the client deployment process 1400 proceeds to step 1450.

If there are no entries in the client/target deployment queue 1380 for the respective client node 835, the process 1400 proceeds to step 1440. In step 1440, the client deployment process 1400 waits for a message to appear in the client/target deployment queue 1380. The client deployment process waits a predetermined amount of time before checking the client/target deployment queue 1380. In a preferred embodiment, the predetermined amount of time can be calculated by determining the next package that needs to be delivered.

In an alternative preferred embodiment, the client deployment process 1400 would be given a hint as to how long the agent should wait 1440 before polling the CDS/ADS for the status of the client/target deployment queue 1380. The hint could be algorithmically derived based on the performance requirements of the application that is being deployed.

In Step 1450, the CDA 944 requests an asset descriptor manifest (ADM) 1452 from the CDS/ADS 965. See Figure 14A. The asset descriptor manifest 1452 is a data structure that associates asset Ids 1454 and offsets 1456 and is used in a preferred embodiment. The ADM 1452 has records 1453 that contain an asset id field 1454, an offset field 1456, and, in a preferred embodiment, an asset type field 1458.

The asset descriptor manifest (ADM) 1452 is used by the CDA 944 to communicate with the CDS/ADS 965 about which assets 1012 or fragments of assets 1012 are yet to be transferred from the CDS/ADS 965 to the respective client node (835, 940) via the CDA (distribution tier) 944.

See Figure 14A. The Asset Descriptor Manifest (ADM) 1490 is made of a list of Asset Manifest Entries (AME, one per asset) 1453. A preferred AME 1453 has the following fields:

- Offset 1456: offset into file containing cached asset, e.g. the "boundaries of the asset fragment".
- Cache name 1478: name of cache file containing the asset.
- Asset_Id 1454: uniquely identifies an asset.
- Timestamp 1479: set to asset timestamp (or version).
- Asset Type 1458: In a preferred embodiment, this field is not included in the ADM 1490 but the information is obtained from data structure 1170 field 1178.

The following describes how these fields are used. The CDA requests an ADM from the CDS/ADS when it finds an entry for it in the client/target deployment queue 1380. The CDS/ADS initializes the ADM with a list of AMEs, one per asset to be deployed to the client. Each AME is initialized as follows:

- Offset = 0
- Cache name = name of cache file containing the asset.
- Asset_Id = asset ID.
- Timestamp: set to asset timestamp (or version).

The CDA then starts requesting the archive containing the assets. The cache name, timestamp, and asset Id field in the AMEs don't change. The "offset" field reflects the offset into the cache file containing the asset. When all of the asset has been transferred, the offset for its AME is set to –1. When all assets on the ADM have been transferred, all AMEs should have a –1 in their offset field.

In a preferred embodiment, the CDA doesn't interpret any fields in the AME and the AME is used by the CDS/ADS only. The CDA monitors the entries in the archive file it gets from the CDS/ADS. An asset may spread across several consecutive calls to the CDS/ADS. When the CDA detects that an archive entry name has changed, then it knows that an asset transfer has completed. The clients keeps track of when archive entries change to be able to reconstruct the archive on the client side. However, in a preferred embodiment, the assets are not deployed until the complete archive has been transferred to the client. Once the client detects that the archive transferred has been completed, it deploys the assets.

In a preferred embodiment, the process uses the views below. They are similar to those views used to determine what assets 240 need to be cached (this is, brought over from the EIS), described elsewhere in this disclosure. However, the views below are used to determine what assets need to be deployed.

VIEW_S3A returns client assets that are up to date with respect to ASSETCACHE table.

CREATE VIEW VIEW_S3A ( CLIENT_ID, ASSET_ID, TIMESTAMP ) AS SELECT CLIENTASSETS.CLIENT_ID, CLIENTASSETS.ASSET_ID, CLIENTASSETS.TIMESTAMP FROM ASSETCACHE JOIN CLIENTASSETS ON CLIENTASSETS.ASSET_ID = ASSETCACHE.ASSET_ID WHERE CLIENTASSETS.TIMESTAMP = ASSETCACHE.TIMESTAMP AND CLIENTASSETS.ASSET_ID IN SELECT ASSET_ID FROM VIEW_S6 WHERE CLIENTASSETS.ASSET_ID = VIEW_S6.ASSET_ID

VIEW_S7A is similar to VIEW_S7, but with respect to ASSETCACHE: it returns a list of assets to be deployed per client.

CREATE VIEW VIEW_S7A ( CLIENT_ID, ASSET_ID ) AS
SELECT CLIENT_ID, ASSET_ID FROM VIEW_S6 WHERE ASSET_ID NOT IN (SELECT ASSET_ID FROM VIEW_S3A WHERE VIEW_S3A.CLIENT_ID = VIEW_S6.CLIENT_ID) AND ASSET_ID IN SELECT ASSET_ID FROM VIEW_S2)

VIEW_S9A is similar to VIEW_S9 but with respect to ASSETCACHE table: it returns list of assets to be deployed per client (with respect to ASSETCACHE table with extra information about cache jar.

CREATE VIEW VIEW_S9A ( CLIENT_ID, ASSET_ID, JAR, TIMESTAMP ) AS

SELECT VIEW_S7A.CLIENT_ID, VIEW_S7A.ASSET_ID, ASSETCACHE.JAR, ASSETCACHE.TIMESTAMP FROM ASSETCACHE, VIEW_S7A WHERE ASSETCACHE.ASSET_ID = VIEW_S7A.ASSET_ID

In Step 1460, the CDS/ADS 965 performs the set operations on the CDS/ADS data store tables (see below in this flow chart description) that are needed to determine which assets (240, 1012) need to be sent to the CDA 944. These set of operations involve determining which assets are registered (in the CLIENTREGISTRY table) [number] for the client id 1382, yet are not already on the client node (835, 940). These asset ids are assembled (by inserting them into a list) into an ADM and returned to the CDA 944 as step 1465.

In step 1470, the CDA 944 calls the CDS/ADS 965 to get the next asset, set of assets, or asset fragment from the CDS/ADS. The specification of the required assets is achieved by sending the ADM 1452 to the CDS/ADS 965 along with request for assets.

In some cases the asset is very large and can spread across more than one client/target 940 call to the CDS/ADS 965. In these cases, the CDS/ADS will send only part of the asset, i.e. an asset fragment to the client/target. The CDS/ADS marks the manifest so that the exact fragments sent to the client/target are precisely known. In this way, the CDS/ADS knows which fragments of the asset it needs to send to the client. In some preferred embodiments, the client/target (CDA) keeps track of the asset ID on the fragment, typically provided on a header. When the CDA detects a change in the asset ID, the CDA knows that a new asset is beginning. In some embodiments, the asset is deployed (placed in the appropriate client/target environment depending on the asset type) as soon as the entire asset is received by the client/target. In other embodiments, the client/targets waits for all the assets on the manifest to arrive at the client/target before deploying all these assets at one time.

In step 1475, the CDS/ADS examines the ADM and returns a data structure to the CDA. The examination of the ADM involves finding the next asset entry for which offset = 0. The CDS/ADS then looks up the asset id 1454 in the CDS/ADS database (e.g. cache) to find where the asset resides in the CDS/ADS asset cache. The CDS/ADS then reads the asset from the cache and builds a data structure for transferring the asset to the CDA. If the asset is large, the CDS/ADS may only read the asset partially, recording an offset 1456 indicating the position the CDS/ADS was at when it completed, in this way, the CDS/ADS could continue sending the asset from the offset where it stopped. If there are more assets, the CDS/ADS repeats the reading and recording process until it has read a sufficient number of assets, either meeting the order specified in the manifest or until the data structure is of a sufficiently large size. The CDS/ADS then proceeds to send the data structure containing the manifest and assets (and/or asset fragments) to the CDA.

In step 1480, the CDA receives the data structure that is the result of the request in step 1475, and proceeds to interpret the data structure. The CDA examines the ADM for the next entry that does not have an offset of –1. If a non-zero entry is the result, the client writes the asset associated with the asset in the entry to local storage and runs the deploy asset adapter

corresponding to the asset type indicated in the ADM entry for the current asset. See the description of Figure 17 below. Note that this way of determining offset is one preferred embodiment and that there are many ways of performing this function known to those skilled in the art having the benefit of this disclosure.

If there are entries remaining in the manifest that have an offset of zero 1485, the CDA proceeds to step 1470.

The reference numbers in the following SQL create statement correspond to the reference numbers in various locations in this description:

```
CREATE TABLE ASSETCACHE (
       ASSET_ID          BIGINT          NOT NULL, 1372, 1172, 1194, 1454
       TIMESTAMP         TIMESTAMP       NOT NULL, 1374, 1179, 1479
       JAR               VARCHAR( 100 )  NOT NULL,
       PRIMARY KEY (ASSET_ID)
```

The primary key in the above tables are a well known way of defining how a table is stored and accessed.

For the purpose of this embodiment, two relational database views, VIEW_S9, VIEW_S7, VIEW_S5A, VIEW_S3, VIEW_S6 and VIEW_S4, are created. Step 1460 uses these views to determine which assets are candidates for client deployment. These views are described as follows:

```
CREATE VIEW VIEW_S9 ( CLIENT_ID, ASSET_ID, JAR ) AS
       SELECT VIEW_S7.CLIENT_ID, VIEW_S7.ASSET_ID, ASSETCACHE.JAR FROM
       ASSETCACHE, VIEW_S7 WHERE ASSETCACHE.ASSET_ID = VIEW_S7.ASSET_ID

CREATE VIEW VIEW_S7 ( CLIENT_ID, ASSET_ID, ACTION ) AS
       SELECT CLIENT_ID, ASSET_ID, 'ADD' ACTION
       FROM VIEW_S6
          WHERE
          ASSET_ID NOT IN
          (SELECT ASSET_ID
             FROM VIEW_S3
             WHERE VIEW_S3.CLIENT_ID =
                   VIEW_S6.CLIENT_ID)
          AND                                  ASSET_ID IN
          (SELECT ASSET_ID
             FROM VIEW_S5A)

CREATE VIEW VIEW_S5A ( ASSET_ID, TIMESTAMP ) AS
       SELECT ASSET_ID, TIMESTAMP FROM
       DEPLOYABLEASSETS WHERE ASSET_ID NOT IN
       (SELECT ASSET_ID FROM VIEW_S4)

CREATE VIEW VIEW_S3 ( CLIENT_ID, ASSET_ID, TIMESTAMP ) AS
       SELECT CLIENTASSETS.CLIENT_ID, CLIENTASSETS.ASSET_ID,
       CLIENTASSETS.TIMESTAMP FROM DEPLOYABLEASSETS
       JOIN CLIENTASSETS ON
       CLIENTASSETS.ASSET_ID = DEPLOYABLEASSETS.ASSET_ID
       WHERE CLIENTASSETS.TIMESTAMP = DEPLOYABLEASSETS.TIMESTAMP
          AND CLIENTASSETS.ASSET_ID IN
            (SELECT ASSET_ID FROM VIEW_S6 WHERE
               CLIENTASSETS.ASSET_ID = VIEW_S6.ASSET_ID)

CREATE VIEW VIEW_S6 ( ASSET_ID, CLIENT_ID ) AS
       SELECT DISTINCT ASSET_ID, CLIENT_ID FROM PACKAGEASSETS JOIN
       CLIENTREGISTRY ON
       PACKAGEASSETS.PACKAGE_ID = CLIENTREGISTRY.PACKAGE_ID
```

```
CREATE VIEW VIEW_S4 ( ASSET_ID, TIMESTAMP ) AS
       SELECT ASSET_ID, TIMESTAMP FROM VIEW_S3
       GROUP BY ASSET_ID, TIMESTAMP
       HAVING COUNT( ASSET_ID ) =
       ( SELECT COUNT( DISTINCT CLIENT_ID ) FROM VIEW_S6
         WHERE view_s6.asset_id = view_s3.asset_id )
```

VIEW_S6 is used to determine the asset ids that the client is registered to receive during client deployment. VIEW_S6 determines the assets id's from the CLIENTREGISTRY table whose package ids match those in the PACKAGEASSET table.

VIEW_S3 is the embodiment of the set operations that determine which assets are up to date. The asset ids specified in the DEPLOYABLEASSET table are intersected using asset id with the asset ids and matching timestamp in the CLIENTASSETS table. The result of this intersection is then intersected with the CLIENTASSET table assets ids that are also contained in the VIEW_S6 view.

VIEW_S4 is the embodiment of the set operations that determine the assets that will not be delivered to any client. The records from the VIEW_S3 view are grouped by distinct asset ids and timestamps. The resulting values are counted and compared for equality with the count of distinct client ids in VIEW_S6 that have matching asset ids in VIEW_S3.

VIEW_S5A is the embodiment of the set operations that determine the deployable assets that are not included in the asset cache. Records from the DEPLOYABLEASSETS table are retrieved based on their asset ids not being found in the VIEW_S4 view.

VIEW_S7 is the embodiment of the set operations that determine the assets that need to be deployed to each client based on client id. Records from the VIEW_S6 view are selected based on their asset ids that are not found in an intersection of the records found in VIEW_S3 whose client ids are also in the VIEW_S6 view. The results are then intersected with the VIEW_S6 asset ids that are in the VIEW_S5A view.

VIEW_S9 is the embodiment of the set operations that determine which assets are contained in the asset cache on the CDS/ADS. Records in the VIEW_S7 view are selected based on whether their asset ids match those in the ASSETCACHE table.

When step 1460 performs the select operation on the VIEW_S9 view, the dependency chain implicit between the VIEWS is traversed creating a single set operation when executed in an optimized relational database management system.

In an alternate embodiment, the SQL queries used to define the views could be combined into a single SQL query. Such a query could be executed to obtain the results indicating which assets need to be added to the ADM.

Figure 15 is a flow chart of the node registration process (NRP) 1500. See also Figure 15A which is a block diagram of a node registration specification data structure 1522. The NRP 1500 covers the specification of the node registration specification data structure 1522 and the

committing of this node registration specification data structure 1522 to the data store on the CDS/ADS.

In step 1510, the node registration specification (NRS) data structure 1522 is specified through either manual or automated processes. The NRS is comprised of a node id 1524 and a package id 1526. The structure 1522 represents the packages that should be delivered to a particular node. For example, this(ese) package(s) would represent the package(s) needed to provide the client/target 940 one or more goods (e.g. movies) and/or services (e.g. financial account) information that the client/target 940 expects (e.g. signs up for, subscribes to, etc.).

In a preferred embodiment, the NRS data structure 1522 is specified as an XML document.

In step 1520, the NRS data structure 1522 is placed on the specification message queue 1205 on the CDS/ADS.

In step 1530, the CDS/ADS 965 reads the NRS data structure 1522 from the message queue.

In step 1540, the CDS/ADS 965 creates a record in the CLIENTREGISTRY table below.

The reference numbers in the following SQL create statement correspond to the reference numbers in Figure 15A:

```
CREATE TABLE CLIENTREGISTRY ( 1522
       CLIENT_ID         VARCHAR( 32 )  NOT NULL, 1524
       PACKAGE_ID        VARCHAR( 32 )  NOT NULL, 1526
       PACKAGE_STATUS    VARCHAR( 32 )  NOT NULL,
       PRIMARY KEY (CLIENT_ID,PACKAGE_ID)
```

The primary key used in the above tables is a well known way of defining how a table is stored and accessed.

In the context of the CDS/ADS, the term adapter refers to a "component-ized" utility program that provides some standardized interface for any given asset type. In this way, the adapter can allow the CDS/ADS to handle any asset type that needs to be distributed over the network. Examples of novel adapters used with this invention include: client deployment 1400, export 1600, version 1660, client deployment 1700, process 1800, target 1900, synchronize 2000, discovery 2100, adjustment 2100A, publishing 2200, and subscriber 2300.

Figure 16A is a flow chart of the version asset adapter process/method (VAM) 1600A. The VAM defines the process of generating version information for an assets based on the asset type 240T. Note that the VAM may need to retrieve version information in different ways based on the type of asset. In a preferred embodiment, the VAM receives a request 1661 for an asset version that typically comprises an asset name, an asset type, an argument string, and a version, e.g. a timestamp. The VAM then calls a Version Asset Adapter (VAA) 1662 associated with the asset type. The VAA then provides current version information of the asset (named by the asset name) that conforms to the versioning convention of the version in the request.

For example, if the versioning convention of the asset type in the EIS is a timestamp like the version convention of the request, the VAA looks for the most current version of the asset (asset name) in the EIS and returns 1664 the current timestamp for that respective asset. For instance, if the asset type is generally associated with an individual file, the timestamp of the file is used to represent the version of the asset.

However, if the asset (having the asset name) in the EIS does not follow the same version convention, e.g. does not use a timestamp for version information when a timestamp is in the request, the VAA has to resolve 1663 the difference by converting or determining indirectly the EIS version information into the convention of the version in the request.

For example, if the asset type 240T is data, a timestamp is not used in the EIS and must be converted 1663 by the VAA based on some other versioning scheme. In a preferred embodiment, the VAA generates a checksum for the data that needs to be extracted. Note that the asset was previously provided with a check sum that was used in the version information in the request. Therefore, the VAA can compare these two check sums to determine if changes have occurred in the data. Thus, check sum information is used as a "timestamp" to determine the most current version of the data asset. Other techniques used for versioning are envisioned.

Specifically, the EIS VAA can perform a query and write the results to a data structure 1664. A simple checksum can be run on the resulting data as if it were a contiguous stream of bytes. The checksum can be sent to and stored on the CDS/ADS. Subsequently, when the CDS/ADS queries the VAA for version information, the VAA can use the checksum to determine if the data has changed. If the data has not changed, the older timestamp is returned or some other indication is made that the data has not changed. Otherwise, if the data has changed, a new timestamp is returned. Thus, the returned timestamp is changed or not depending on whether the checksum has changed or not. In this manner, a timestamp is used to determine the most current version of the data, even though another means, e.g. the check sum, is really used to determine the version of the data.

Figure 16 is a flow chart of the EIS export asset adapter process/method (EAM) 1600. The EAM defines the process of creating a generalized data structure from any number of assets based on the asset type. In a preferred embodiment, the data structure 240 is that of Figure 2 described above, and will be referred to as such in the description below without loss of generality.

The process selects the appropriate export asset adapter (EAA), the implementation specific construct for this adapter method, based on the asset type 240T the adapter will create a data structure 240 appropriate for distribution. The data structure will depend on the asset type. Examples of these structures are described in the table above in the description of Figure 2.

In step 1605, the EIS deployment agent (EDA) determines if the asset is any one of the following types: static content, JSP (Java Server Page), or Java Bean. This determination is

done by looking at the asset type field 1178 in the asset specification 1170 data that was sent with the request for exportation. If the asset is one of the above types, the EDA tasks the appropriate EAA to perform step 1610. Otherwise, the EDA proceeds to step 1615. In a preferred embodiment, the asset type 1178 is provided with the request for exporting the respective asset that initiates the EAM.

In step 1610, the EAA copies the file specified by the asset specification. The EDA then copies the file into an appropriate data structure and proceeds to step 1645. Other asset types 240T that can be processed by copying without an extended environment also would be handled this way.

In step 1615, the EDA determines if the asset type is either SB (Session Bean) or EB (Entity Bean). If the asset is one of these types, the EDA tasks the appropriate EAA to perform step 1625. Otherwise, the EDA proceeds to step 1620. In a preferred embodiment, asset types are defined in any of the package specifications (e.g. 1190, 1100); therefore, the CDS/ADS specifies the appropriate asset type in the request for the EIS assets.

In step 1620, the EDA determines if the asset type 1178 is either RD (Reference Data) or ED (Entity Data). If the asset is one of these types, the EDA tasks the appropriate EAA to perform step 1635.

In step 1625, the EAA extracts implementation class files for an EJB, and then extracts the EJB deployment descriptor and the stubs and skeletons (J2EE/EJB deployment descriptor and the stubs and skeletons well know part of the EJB specification) from an archive specified by the asset specification 1170. The EDA then copies the EJB implementation class file, as specified by the asset name field 1176 and dependent on the EJB implementation on the EIS node, into the LD 210 of the asset data structure 240. EDA copies the J2EE/EJBdeployment descriptor and stubs and skeletons into the extended environment part 220 of the asset data structure 240. The EAA then proceeds to step 1630.

In step 1630, the EAA determines if the asset type is SB, if so, the EAA proceeds to step 1635, otherwise, the EAA proceeds to step 1645. The type 240T is determined by examining the specification.

In step 1635, the EAA determines if the asset specification 1170 has a specification of a "where clause". The "where clause" is a string that specifies information that will be used to perform an SQL query. If a "where clause" is specified the EAA proceeds to step 1640, otherwise the EAA proceeds to step 1650. Where clauses are well known in SQL.

In step 1640, the EAA selects data from a table in a database on the EIS node. The table for the selection is specified in the asset specification and the "where clause" for specifying a subset of the data is from step 1635. The EAA places the records resulting from the database record selection in the respective LD 210.

In step 1645, the EDA takes the data structure 240 created in the proceeding step and sends the data structure 240 to the CDS/ADS.

In step 1650, the EAA selects all the records from the table specified in the asset specification 1170 (Figure 11A). Then the EAA proceeds to step 1645.

Figure 16B is a flow chart of an alternative preferred EIS export adaptor process.

While process 1600 creates a package data structure (above), in an alternative embodiment, the export adaptor process 1600B creates 1650B one or more preliminary package specifications 1100A of one or more digital assets. In this preferred embodiment, a packaging adaptor process 1600C is used to create the final package of digital assets 240.

The process 1600B starts by traversing an intermediate representation 2100C (below) of one or more parts 100F of a computer system created by the discovery process while applying one or more context rules 1610B to determine a context of the parts 100F. Processes for traversing graphs are well known. For example, see well-known works by Prim, Kruskal, Dijkstra, et. al. In step 1615B, the process 1600B determines if the context is a standard specified context.

A standard specified context describes a common topological deployment pattern of EIS files and data throughout the file systems of a computer or network of computers. In addition, a context contains other rules and information (described below). Deployment patterns covering the files and data for many computer system parts 100F, and particularly, for most complex business application software products such as EIS application servers, EIS web servers, DBMS products, CRM products, etc, have become standardized over time due to the formal recommendations of standards bodies, through popular acceptance become industry custom, or through the stated requirements of the providers of such system part or software.

For example, when the Sun Microsystems product Java Web server is installed, a specific directory on the computer must be identified as the directory where HTML files are placed. Through custom, this directory has been named "public_html". Other, later Web and application server software products followed this custom, and now, most Web and application server products require or recommend placement of HTML files in a directory named "public_html". Other directory names have become standard locations for specific files or types of files in Web and application servers. These include "cgi-bin" as the location of various script files, and "servlet" for Java language servlet files. Web/application server products which follow one or more of these directory naming customs include Inprise AppServer, BEA's Weblogic application server, and the Tomcat Web server.

More generally, a directory named "lib" is utilized by many complex business software products as a location for callable methods, sub-programs, and functions. Examples of software products that use a "lib" directory for this purpose include IBM Corp's DB2 DBMS, Ghostgum Software's GSView (a product for viewing postscript documents), the Inprise AppServer, Oracle DBMS, and TogetherJ from ObjectInternational, Inc. (a product for authoring, viewing and managing UML diagrams). Other directory names that are in common

use as locations for specific files or types of files are "bin", "users" and "docs". All of the above are examples of common topological deployment patterns of system part 100F files and data, which can be described by a context. A standard specified context describes a common topological deployment pattern of EIS files and data throughout the file systems of a computer or network of computers.

The process 1600B executes a rule-based data-driven analysis (such as that performed by an expert system) of the containment graph or other intermediate representation 2100C of the subject system part 100F to determine the context for the subject system part 100F. This rule following analysis compares the topological deployment pattern of the system part 100F as manifest in the intermediate representation 2100C against a repository of candidate standard specified reference contexts to determine if the topological deployment pattern matches one of the referenced standards in the repository.

Given well-known topological deployment patterns of system part 100F files and data (as described above), such an analysis can identify the context of the subject system part 100F (i.e. Win32 application, J2EE application, BEA Weblogic, etc.) without being provided with an explicit map of the files and data for every individual system part on the EIS. By determining that the topological deployment pattern is one of a standard reference context, the process 1600B is able to locate all the required member objects of the system part 100F without having detailed information (e.g. an explicit map) of the files and data of the subject EIS system part.

Once a context is determined, and the context is determined to be a referenced, standard specified context, the process will perform a directed search 1620B constrained to context-established locations on the EIS. The term "directed search" indicating a goal-oriented search utilizing a priori knowledge of the search domain is well known. However, the directed search performed by the export process 1600B utilizes the context to constrain searches to specific EIS directories and is considered novel

The other information contained in a context (as mentioned above) is in addition to the description of a topological deployment pattern. This other information specifies whether 1625B an actual search 1635B of the specific EIS (as above) directories is required. For example, the software Java language decompiling program SourceAgain (a product of Ahpah Software), requires that only a single runnable member object, "srcagain.exe" be deployed for proper use of the product. No search of the EIS file systems for other runnable member objects need be performed in order to utilize the software, once the "srcagain.exe" member object has been acquired.

Alternatively, some system parts 100F might require a search 1635B for runnable member objects. An example of this case would be a system part with a plurality of dynamic link libraries (DLLs), where only certain of the DLLs are of interest. The limitation of interest is due to that fact that generally, individual DLLs provide discrete functionality. For instance, one DLL may provide only the functionality required for local printing, while another DLL may provide the functionality required for wireless communication. If the purpose of the

discovery and export processes is to enable wireless communication, the DLL to enable local printing is not of interest. Therefore, a rule would identify only the wireless communication DLL. Specifically, within the context of the system part 100F the wireless communication DLL would have unique features that would be used to apply the rule during the search 1635B.

Apart from determining the context of the system part 100F and apart from the context itself, the process 1600B uses a separate set of rules in order to identify runnable member objects in the locations of EIS specified in 1620B.

Once the search 1635B for runnable member objects is complete, and the runnable member objects are identified, an identifier consisting of the asset id 1172 corresponding to the respective runnable member object (member object by member object) is placed in a preliminary package specification 1100A. In one preferred embodiment is a simple list, and the asset id is used to access the respective digital asset 240 in the Asset Inventory 2150B created by the discovery process 2100B (below).

When the process 1600B accesses the digital asset 240 in the asset inventory 2150B, the process 1600B updates the extended environment 220 of the respective digital asset 240 as described in detail below, the updating done by adding one or more export descriptors to an extended environment 220 of the respective updated digital assets.

If it is determined 1625B that a runnable search is not required, it is then determined whether 1640B a search 1645B for non-runnable member objects needs to be performed. If, by the context, it is determined that only one (or a fix number) of identified non-runnable member object constitutes all non-runnable member objects of the system part 100F, no search 1640B needs to be performed. Identifiers for these identified non-runnable member objects are then placed 1650B in the preliminary package specifications 1100A and the process 1600B performs any other updates to the preliminary package specification 1100A required.

Alternatively, some system parts 100F might require a search 1645B for non-runnable member objects. An example of this case would be a system part with a plurality of image files (e.g, .GIF files), where only certain of the image files are of interest. The limitation of interest is due to that fact that generally, image files are used by specific function areas of a system part 100F. For instance, in the SourceAgain application 108 mentioned above, there are GIF files supporting product and sales HTML pages. These product and sales HTML pages might not be of interest in user applications that solely require de-compilation. Therefore, rules for selection would exclude these non-runnable member objects from selection.

In alternative embodiment, the asset id (for runnable and/or non-runnable member objects) is used to access and update information about the respective digital asset 240 from the asset definition data structure 1170. In this embodiment, the process 1600B can provide process 1600 with the preliminary package specification listing 1100A, to the table in step 1650 of process 1600 (above).

In an alternative preferred embodiment, the process 1600B can determine that the system part 100F has no standard context 1615B. If it is determined that no standard context exists, an implicit traversal of the EIS files and data structures must be performed 1630B. Implicit traversal 1630B requires traversal of the entire intermediate representation 2100C as opposed to the directed search 1620B.

In this alternate preferred embodiment, the search for runnables 1635B that is performed during the implicit traversal of the intermediate representation utilizes a number of well-known techniques. The most well known technique is examination of member object file extensions. Generally, those member objects that meet the definition of a runnable (as described above) have file extensions that are named using a well known finite set of file extensions. For example, in a Windows NT 4.0 environment, ".EXE", ".JS", ".DLL", and ".CLASS" are common and well known file extensions for (respectively) executable, java script, dynamic link library, and java class runnable member objects. Likewise, in an Sun Microsystems Solaris 8.0 environment, a ".SO" file extension indicates that the file is a runnable member object. An example rule for the search 1635B Windows NT 4.0 environment would be that if a file extension is ".EXE" then select the file as a runnable member object. In one preferred embodiment, all files that are not runnable member objects are treated as non-runnable member objects 1645B.

During the execution of the searches for runnable 1635B and non-runnable 1645B member objects during the implicit traversal 1630B, the identification of each newly identified member object is followed by a new attempt to determine a context for the subject system part 100F. Rule sets are examined and sorted in the form of a diagnostic or pattern-matching algorithm similar to the rule-based, data-driven operation of an expert system. At any time the process has sufficient confidence in a conclusion, the implicit traversal is suspended, and an execution of a directed search 1620B using the candidate context is performed. If the results of that directed search 1620B are successful, the implicit traversal 1630B is abandoned. If the directed search 1620B fails, the implicit traversal 1630B is resumed. If the implicit traversal of the intermediate representation 2100C performs a complete traversal of the implicit representation without identifying a matching context, the implicit traversal is considered a failure, and a human deployment engineer may be needed.

In these cases, the human deployment engineer could develop additional rules to place in the expert system's rulebase so that future invocations of the expert system would succeed in locating all files and completing all searches called for in the rulebase. Although the use of an expert system for export of assets is believed to be new, expert system technology is well known in the art.

As before, the preliminary package specification 1100A is updated 1650 each time a runnable and/or non-runnable member object is identified by the process 1600B. In alternative embodiments, the asset inventory and asset definition data structure 1170 are also updated as required and described above.

The extended environments 220 of the digital assets 240 corresponding to the respective selected runnable and /or non-runnable member objects are also updated 2150B.

At the end 1685B of the process 1600B, an updated asset inventory 1665B with updated extended environments 220 is completed for the system part 100F. In alternative embodiments, an updated asset data structure 1660B.

A description of alternative ways to update the EE 220 is now presented.

In one preferred embodiment, the descriptors include one or more common descriptors that provide a unique identification of the digit asset on the networks. These common descriptors include one or more of the following: a common security export descriptor and a common runnable descriptor.

In one preferred embodiment, the export descriptors include one or more dependency descriptors. The dependency descriptor includes one or more asset dependency descriptors.

In one preferred embodiment, the dependency descriptor includes one or more EIS server dependency descriptors. These EIS server dependencies may include one or more of the following: EIS database management systems (DBMS), EIS servers, EIS application servers, EIS web application servers, one or more business applications, and one or more accounting application. The EIS server dependencies also may include one or more of the following: one or more Oracle DBMS, one or more Sybase DBMS, and one or more DB2 DBMS.

In a preferred embodiment, the export descriptors include one or more type descriptors. The type descriptors may include any one or more of the following examples: static content (SC), dynamic content (DC), enterprise java beans (EJB), reference data (RD), session bean (SB), entity bean (EB), entity data (ED), java class (JC), Java Connector Framework connectors (JCF), Java applet (JA), and java beans (JB).

In one preferred embodiment, the export descriptors include one or more asset category descriptors. These asset category descriptors may include any one or more of the following examples: a content descriptor, a presentational descriptor, a transactional descriptor, and a relational data descriptor.

In one preferred embodiment, the descriptors comprising one or more asset class descriptors. The asset class descriptors include any one or more of the following examples: base, java, non-java, language, and non language.

In one preferred embodiment, the descriptors comprise one or more schema descriptors. The schema descriptors provide information that describe any or more of the following examples: database table names and definitions, database column names and definitions, database key identifiers and value ranges, database view names and definitions, and other well known database schema elements.

In one preferred embodiment, the descriptors further comprise one or more metadata descriptors. The metadata descriptors provide information that describe any or more of the following examples: repository object definitions, scope object definitions, module object definitions, operation object definitions, exception object definitions, constant object definitions, properties object definitions, attribute object definitions, relationship object definitions, type object definitions, and other well known metadata object definitions.

In one preferred embodiment, the descriptors comprise one or more transform descriptors. The transform descriptor may describe a transformation of data in a logic/data section of the digital asset. The transform descriptor may also describe a transformation of logic in a logic/data section of the digital asset. The transform descriptor may include the following:

a properties descriptor that provides information required for use of the digital asset on an operating system of the base execution environment.

a format descriptor that provides information required for use of the digital asset on an operating system of the base execution environment.

a registry descriptor that provide information required for use of the digital asset on a Window's operating system on the base execution environment.

In one preferred embodiment, the descriptor comprises one or more reference descriptors. The reference descriptor may include any one or more of the following: a reference link descriptor, a reference file descriptor, and a reference directory descriptor. The reference link descriptor may provide a world-wide-web (WWW) address that has contents used for processing of the digital asset. The reference link descriptor may also provide a world-wide-web (WWW) address that has contents used during execution of the digital asset. The reference file descriptor is a unique fully qualified name of a file required for reference by the digital asset. The reference directory descriptor may provide additional address information that is used to locate one or more of the associated digital assets.

In one preferred embodiment, the descriptors comprise one or more security descriptors. The security descriptor may include any one or more of the following functions: encryption, authorization, and access control.

In one preferred embodiment, the descriptor comprises one or more package relationship descriptors that represent a part-whole relationship between the digital asset and one or more packages containing the digital asset. The package relationship descriptor may represent at least the following three relationships in the part-whole relationship: a mandatory part-whole relationship, a shared part-whole relationship, and a root part-whole relationship.

In one preferred embodiment, the descriptor comprises one or more distribution logic descriptors, each having one or more transactions rules and one or more concurrency rules. The transactions rules specify any of a number and a frequency of times that the digital asset can be distributed to one or more target computers. The concurrency rules specify whether or

not there are any restrictions on distribution of the digital asset with respect to the distribution of one or more other digital asset.

Figure 17 is the flow chart of the client deployment asset adapter (DAM) method/process 1700. The DAM 1700 defines the process for taking the data structure 240 created by another asset adapter method (e.g., export 1600, process 1800, or target 1900) and adjusting the local environment based on the extended environment 220 of the respective data structure 240. The data structure 240 provides the extended environment 220. The Client Distribution Agent (CDA) selects an appropriate client deployment asset adapter (DAA), the implementation specific construct for the DAM, based on the respective asset type 240T being deployed. The CDA tasks the DAM with creating the necessary changes to the local (i.e., target 940) environment to allow for the asset 240 to be deployed and to execute properly. In other words, the DAM 1700 deploys the asset 240 on the respective client environment by changing and/or adding to the client environment, if needed, as determined by information in the extended environment 220 of the asset 240.

In a preferred embodiment, in step 1705, the CDA 944 use the DAM 1700 to determine if the asset is any one of the following types: static content, JSP (Java Server Page), or Java Bean. If the asset is one of these types, the CDA tasks the appropriate DAA to perform step 1745. Otherwise, the CDA proceeds to step 1710.

In step 1710, the CDA determines if the asset type is either SB (Session Bean) or EB (Entity Bean). If the asset is one of these types, the CDA tasks the appropriate DAA to perform step 1715. Otherwise, the CDA proceeds to step 1740.

In step 1715, the DAA (associated with SB's or EB's) inserts the EJB classes for the respective SB/EB into the client archive appropriate for the local environment. In other words, the DAA inserts the EJB implementation class (on the respective asset 240) into an archive where the other EJBs are stored on the client, along with the stubs and skeletons of the respective EJB. The DAA also inserts the altered the client deployment descriptor from the extended environment 220 portion of the asset data structure 220 into the client EJB environment. The DAA then proceeds to step 1720.

In step 1720, the DAM determines whether the asset is an EB, if so, the DAM proceeds to step 1725. If not, the DAM process 1700 ends.

In step 1725, the DAM determines if the table specified in the asset logic/data (LD) 210 exists in the local (client) database. If the table exists, the DAM proceeds to step 1735. Otherwise, the DAM proceeds to step 1735.

In step 1730, the DAM creates the table specified in the asset data structure 240, then proceeds to step 1735. In a preferred embodiment, the table specification is contained in the extended environment section 220 of the asset data structure 240.

In step 1735, the DAM imports the data from the data structure 240 into the database table located on the client. In a preferred embodiment, the imported data is extracted from the LD section 210 of the asset 240.

In step 1740, the DAM determines if the asset type is either RD (Reference Data) or ED (Entity Data). If the asset is one of these types, the DAM proceeds to step 1725. In a preferred embodiment, the DAM would also write an entry in a system database table indicating that a certain set of data was written to the database (e.g. a directory reference), permitting any other process to be notified as to what data has been added.

In step 1745, the DAM copies the file specified by the asset 240 from the data structure 240 into the location appropriate for that asset type in the file system. Each asset adapter was configured to know where to store the assets of certain asset types.

After the deploy adapter method is run for a given asset 240, the particular target 940 has incorporated the necessary data, interfaces, and/or control functions to enable the asset purpose on the respective target 940.

Figure 17A is a block diagram of one preferred J2EE transactional deployment sphere of control 1700A.

At any given node on the network a transaction deployment of assets can be enabled. In one preferred embodiment, the transaction deployment applies to J2EE assets, i.e. components. Figure 1700A shows any given network node with the J2EE transactional deployment sphere of control 1700A.

The node contains a well-known J2EE application server 1705A. The J2EE application server contains one or more well-known J2EE applications 1710A. The J2EE application has a web container 1720A and possibly an Enterprise Java Bean (EJB) container 1725A that are also well-known and referred to as J2EE application containers. Note that the J2EE standard specifies a J2EE deployment transactional sphere of control 1715A that encompasses only the web container 1720A and the EJB container 1725A. Further note that the J2EE standard does not explicitly apply transactional semantics to the deployment into a web container or a transactional container (EJB container). For more complete description, see "Designing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition" Nicholas Kassem and the Enterprise Team Version 1.0.1 Final Release October 3, 2000 which is herein incorporated by reference in its entirety.

The web container 1720A can contain static content, Java Servlets, Java Server Pages (JSP), HTML pages, and XML pages. The EJB container can contain enterprise beans. All of these can be referred to as container components.

The CDS/ADS can deliver and deploy container components, called delivered J2EE components to the respective web container 1720A or EJB container 1725A. Delivered J2EE components include delivered J2EE web components 1730A, delivered J2EE EJB

components 1735A, and delivered J2EE application data 1740A. These delivered components 1732A are well-known and part of the specification incorporated above.

In this preferred embodiment, system parts 100F are deployed via under the control of the CDS/ADS to the J2EE application server 1705A and any participating database 1750A, typically, on one or more EISs or other sources. Therefore, there is a logical connection 1745A through the CDS/ADS by which J2EE application data 1740A is migrated from the original source of the data to the database 1750A.

In this preferred embodiment, a novel sphere of control 1776A encompasses the J2EE application server 1705A, the J2EE applications 1710A, the J2EE application containers (1720A and 1725A), the web container, the EJB container, the J2EE application container components (not shown), the delivered J2EE components (1730A, 1735A, and 1740A, typically 1732A), and the logical connections 1745A. The sphere of control 1776A manages a transactional deployment of the delivered J2EE components 1732A and an update of the databases 1750A to have data consistent with the J2EE application 1710A. While spheres of control are well-known in the prior art, the scope of this sphere of control 1776A encompassing the database 1750A as well as the J2EE application 1710A and other aspect defined here, is considered novel. See Davies, C. T., "Data Processing Spheres of Control", IBM Systems Journal 17(2): pages 179-198, which are herein incorporated by reference in their entirety.

Figure 17B is a flow chart of the implementation process 1700B of one preferred J2EE transactional deployment within the sphere of control 1776A of Figure 17A.

The process 1700B begins with the CDS/ADS having acquired and delivered the following components in any combination: J2EE EJB components, J2EE web components, and/or J2EE application data.

Step 1701B tests if database only information is received or not. If only database information is received, the process 1700B goes to step 1750B.

Step 1703B tests if only J2EE components are received or not. If only J2EE components are received, the process 1700B goes to step 1725.

If both 1705B database information and J2EE components are received, the process goes to both steps 1725B and 1750B.

Step 1750B of the process 1700B attempts to open 1750B the database 1750A. Step 1750B also attempts to deploy the J2EE application data 1740A into the database 1750A. If this deployment fails (succeeds), the database 1750A provides the process 1700B notification that the database deployment failed (succeeded).

Step 1725B communicates with the J2EE application server 1705A and requests the server 1705A attempt to deploy the J2EE web component 1730A into web container 1720A. Step

1725B also/alternatively can request the server 1705A attempt to deploy the J2EE EJB components 1735A into the EJB container 1725A. If either or both of these attempts fail or succeeds, the J2EE application server 1705A provides the process 1700B notification of the respective failure or success.

In the event that there is either only a database available to open, or, only one of a J2EE web container or J2EE EJB container to open, the process 1700B handles these events using prior art methods, (not shown in Figure 17B) and returns any result codes from those prior art methods.

Database vendors provide this functionality to report failures and successes in accessing data in their databases 1750A.

While the deployments and notifications are J2EE application server are vendor specific, in general, the deployment and notification of failure or success are defined by the J2EE specification incorporated above.

However, deployment and notification of failure for the database 1750A external to the server 1705A are not disclosed or recognized in the J2EE specification. Reasons for this failure to disclose are based upon the intentionally partitioned design of J2EE applications between presentation, business logic, and external (backend) databases 1750A. For further elaboration of those reasons see page 15 of the specification incorporated above.

Applications deployed using our invention preserve the flexibility in choice of database 1750A and application server 1705A which the J2EE architecture has been designed to provide, but, in addition, provide a level of deployment management above either the application server 1705A or database 1750A alone. This level of deployment management is exercised at the scope of the sphere of control 1776A described in 1700A.

Step 1710B determines if the opening of the database 1750A and the attempt to deploy data into the database 1750A was successful or not.

Step 1715B determines if the opening of the containers (1720A and/or 1725A) and the attempt to deploy the respective J2EE web components and J2EE EJB components into the containers (1720A, 1725A) was successful or not.

In one preferred embodiment, the prior version of the delivered J2EE components 1732A including the J2EE application data 1740A are deleted (disposed of) 1720B only if both the deployment of the J2EE application data 1740 and the deployment of the J2EE web components 1730A and the J2EE EJB components 1735A into the respective web container 1720A and the EJB container 1725A is successful 1730B. In another preferred embodiment, the manifest specifies the exact components/assets to be removed, and the purge operation is performed on those components/assets.

If the opening of the database 1750A and the attempt to deploy that data into the database 1750A is unsuccessful but the deployment of the J2EE web 1730A and EJB 1735A components is successful, the previous application running on the J2EE server 1705A is restored 1742B. This provides a durable and consistent state for the server 1705A.

In a preferred embodiment, the current state of the database and application containers is stored before attempting to deploy. This is done via a combination of the synchronization lifecycle step and export adapters for the container. The synchronization step sends new DBMS updates back to the distribution server, if there is a conflict with backend data, it will be resolved and sent back to the target node. The export of application container state allows for the successful restoration of the application containers without completely reverting to the last distribution.

Optionally, additional actions can be taken that are governed by failure rule execution 1744B.For example, notifications could transmitted to a system administrator, or written to a system log or monitor.

Conversely, if opening the containers (1720A and/or 1725A) and/or the attempt deploy the respective J2EE web 1730A or EJB 1735A component is unsuccessful and the opening and access of the database 1750A to deploy data 1725B is successful 1715B, the database 1750A updates are rolled back 1744B.

Optionally, additional actions can be taken that are governed by failure rule execution 1744B.For example, notifications could transmitted to a system administrator, or written to a system log or monitor.

If both 1731B the opening/deployment 1750B into the database and the opening/deployment 1725B into the containers fail (1710B, 1715B, 1730B), either no action is taken or one or more failure rules are applied 1740B.

Failure rules govern the behavior of the process 1700B in the event of both (1710B, 1715B) failures.

Examples of failure rules include: a retry of the data object deployment, a retry of the delivered J2EE component deployment, a retry of the failing deployment, an accessing of an alternative source for the data of the failed data deployment, and an accessing of an alternative source for the component of the fail component deployment.

Figure 18 is the flow chart of the CDS/ADS process asset adapter method/process (PAM) 1800. The PAM defines the process of creating a version of the asset 240 that is appropriate for the target's (e.g. client) base environment. More specifically, the asset's logic/data portion 210 and/or extended environment 220 is translated in such a way that it can allow the asset to perform its functions in the target's base environment.

For example, the asset 240 may have come from a source's base environment such as a Weblogic EJB. The target's base environment may be OpenEJB (or JBOSS). Therefore, the asset's extended environment would need to be converted from Weblogic to OpenEJB (or JBOSS).

In step 1805, the CDS/ADS determines if the asset type is either SB (Session Bean) or EB (Entity Bean). If the asset is one of these types, the CDS/ADS tasks the appropriate process asset adapter (PAA), the implementation specific construct for this adapter method, to perform step 1810.

The PAA is selected on the basis of both the asset type (LD portion) and the target format/environment. In alternative embodiments, the PAA is selected on the basis of the extended environment (EE) and/or the LD and the target format/environment. In other preferred embodiments the PAA is selected on the basis of descriptors contained in the EE 220.

In step 1810, the PAA translates the deployment descriptor (The word "descriptor" can refer to the descriptors that actually comprise the type of EJB asset 240 discussed in this section and is well known in the EJB specification. These descriptors are explicitly differentiated from the descriptors that comprise the EE 220 and are described in Figure 2B.) into the target (client) format. The EJB deployment descriptor is obtained from the data structure 240 corresponding to the asset id located in the asset cache; more specifically, it is the extended environment 220 portion of the asset. PAA also removes the EJB implementation from the LD 210 portion of the asset 240. The PAA then proceeds to step 1815.

In step 1815, the PAA generates the EJB stubs and skeletons for the target environment. The PAA places the EJBs from the asset cache onto the file system. Then the PAA executes the appropriate utilities to generate the classes that the default EJB environment (at the client) will require to execute the EJB (these are the stubs and skeletons). The PAA then proceeds to step 1820.

In step 1820, the PAA moves the new stubs, skeletons, and translated deployment descriptor back into the asset data structure 240. More specifically, the stubs, skeletons, and translated deployment descriptor are moved into the extended environment 220 section of the asset. The PAA then moves the data structure 240 back into the asset cache.

In a preferred embodiment, the Process Adapter Method Process (PAM) 1800 is performed on one more assets 240 before the assets 240 are cached in the CDS/ADS cache memory. In this embodiment, the process method performs any intermediate processing required. For example, the EJB adapter's process method could create the appropriate EJD proxies, so this processing does not occur at the client.

Figure 19 is a flow chart of the CDS/ADS target asset adapter method (TAM) 1900. The TAM defines the process of creating a targeted data structure from any number of assets 240 based on the asset type 240T and other asset information. The process 1900 selects the

appropriate target asset adapter (TAA), the implementation specific construct for this adapter method, based on the asset type and tasks the adapter with creating a data type appropriate targeted for distribution to a particular client node.

The process method is typically a process that needs to be done an asset for all respective target environments that are registered for the package(s) that contain the respective asset. The target method is processing that is required for an asset for one or more particular target/client that is registered for the respective asset.

In step 1905, the CDS/ADS determines if the asset type is either RD (Reference Data) or ED (Entity Data). If the asset is one of these types, the CDS/ADS tasks the appropriate TAA to perform step 1910.

In step 1910, the TAA retrieves a "where clause" specified in the asset data structure 240, typically in the extended environment 220. The TAA then does a token replacement operation on the "where clause". In a preferred embodiment, the token replacement is a basic string replacement operation, retrieving the replacement value from the node specification corresponding to the targeted node. The TAA proceeds to step 1915.

In step 1915, the TAA runs a query on the table specified in the asset data structure 240 using the "where clause" from step 1910. The TAA proceeds to step 1920.

In step 1920, the CDS/ADS takes the records returned from the TAA in step 1915 and puts the records into a simple data structure. The CDS/ADS then puts the data structure into the LD section 210 of the asset 240 and returns the asset to the asset cache.

For example, an asset of type Reference Data (RD) might need to be targeted for each client node. The RD's "where clause" would have tokens that need to be replaced using information in the node's registry. The asset adapter would perform an SQL query using the transformed where clause. The data resulting from this query would be placed in the LD 210 section of the asset. The asset would then be placed back into the asset cache.

Another example of the TAA would be for the Entity Data (ED) asset type. The asset specification might indicate that a targeted query has to be performed for each target client node. For the case of an eCommerce application, the ED adapter might query the status of the current orders for an individual customer. The data would then be distributed to that client. If the client's name were John Smith, only the order status information would be downloaded to John Smith. He would receive his order status information, and no one else's. Likewise, other clients would not receive John Smith's order status information. The query is specific to John Smith and is performed when the John Smith's client node is ready to receive the information.

In one preferred embodiment, the data resides on the EIS tier. In an alternative preferred embodiment, the data is replicated to the CDS/ADS tier and the targeted query is performed on the CDS/ADS tier as explained in the paragraphs above.

Figure 20 is a flow chart of the synchronize asset adapter method (SAM) 2000. One function of the SAM 2000 is to move a "synchronization asset" from one or more targets to one or more sources. (A synchronization asset 240S is an asset 240 that is moved from a target environment to a source environment.)

The SAM typically defines the process by which one or more source environments (910, 915) can be synchronized with the target environment. However, note that in a general case, synchronization can be used to update information, e.g. changes in assets, on any node 940 in the network - including server nodes or other target nodes. Synchronization can also be used to update assets in the CDS/ADS server (or target and/or EIS nodes performing a distribution function) so that the updated assets in turn can be distributed to other nodes of the network. While, the description below refers to moving information "back" to a source environment in order to update the respective source environment, synchronization can be used to move required information any where on the network to any given network node to provide new information and/or updated information without loss of generality.

Note that in some embodiments, the synchronization process could be the export process 1600 with the source and target roles reversed. However, a preferred embodiment, uses process 2000 for synchronization.

The first step involves the target environment creating a synchronization asset. The synchronization asset is subsequently moved to the CDS/ADS, then onto the source environment. The source environment is synchronized with the synchronization asset. Synchronization means that the source environment will be changed to reflect a change in one or more of the target environments.

In step 2010, the CDA is called by an external process (e.g., in the base environment of one or more targets or any other process external to the CDS/ADS) indicating that a synchronization should happen. The SAM proceeds to step 2020. The call includes argument data that the CDA will pass on to the synchronization asset adapter (SAA) in an hidden fashion. The SAA is an implementation of the SAM 2000 for a particular respective asset type. See step 2020.

For example, the asset specification, e.g. 1175, for an ED type may have argument data that indicates that a synchronization should happen every time that a database record in the table corresponding to the ED is changed. The client DBMS would be notified when the ED was deployed into the target/client environment in which this event needs to occur. The client DBMS would then call the CDA when the database record had changed, passing arguments to indicate the asset type, ED, and arguments that allow the SAA (for the respective ED) to know how to build the synchronization asset (or the change in information).

In step 2020, the CDA selects the appropriate synchronization asset adapter (SAA) for the asset type indicated by the caller of the process. The CDA passes the hidden arguments to the SAA, which proceeds to step 2030.

In step 2030, the SAA determines if the asset type is ED or EB. If not, the method ends, if so, the method proceeds to step 2040.

In step 2040, the SAA determines which table is associated with the asset by examining the arguments. The SAA then retrieves the synchronization information from the client environment for the respective asset. In the case of an ED or EB, the retrieval information constitutes the insertion, deletion, and updating of database records which constitute the changes that the data has undergone at client since being deployed into the client target environment for this respective asset.

In step 2050, the SAA contacts the CDS/ADS, and transfers the asset data structure 240S to the CDS/ADS. For example, this part of the SAA, running on the client environment, can update the LD 210 and/or the EE 220 parts of the respective asset 240 to covert the asset 240 to a synchronization asset 240S.

In step 2060, the CDS/ADS determines which (source) EIS node is associated with the asset. The CDS/ADS does this by retrieving data from the asset specification 1175, e.g. machine location 1174. Note that this step, the SAA (i.e. step 2060) in this example is running on the CDS/ADS.

In step 2070, the CDS/ADS initiates a connection to the appropriate EDA on the EIS (source) node determined in step 2060, and sends the synchronization asset 240S to the EDA/source node.

In step 2080, the EDA examines the asset data structure 240S to determine the asset type. The EDA then selects the appropriate synchronization asset adapter (SAA) 2000 for that asset type. Note that this SAA is running on the source node/EIS.

In step 2090, the SAA determines if the asset type is ED or EB. If not, the method ends, if so, the method proceeds to step 2095.

In step 2095, the SAA applies the synchronization information (e.g. LD 210 and/or EE 220) in the synchronization asset 240S of the source environment. Since this is an ED or EB, the changes involve the insertion, deletion, and updating of database records against the source database for the respective asset.

In one embodiment, the synchronization for data assets, ED & EB, are achieved by recording the SQL queries that are applied to the database table. The queries are recorded in a data structure that is written to a file. The file can then be transferred to the database server node, via one or more CDS/ADS nodes. Once the file is on the (source) EIS node with the database table the data originated, the SQL queries are applied to the database table. If the database table has been partitioned well, the queries will synchronize the sub-partition of the table to the state of the table on the client node.

In alternative embodiments, the source node can be updated with the entire synchronization asset 240S or with just the necessary changes that are to be made to the respective asset.

The routing of the data from the client node to the EIS node is achieved due to the CDS/ADS nodes keeping the routing information for the data. When it comes time to synchronize the data, the CDS/ADS nodes know where the originally received the data came from and the CDS/ADS sends the data back through those respective nodes.

Note that the SAM/SAA 2000 is a process that is typically distributed over several tiers of the network. The SAM/SAA provides processing support for the synchronization asset on its "life cycle" 240L journey back to the source (EIS) node. On this journey the synchronization asset 240S carries information reflecting the changes made to the associated asset deployed on its respective remote client environment. These changes are then updated at the source.

In a preferred embodiment, the SAM/SAA 2000 can combine synchronization assets 240 from multiple targets/clients into a "coalesce synchronization asset" that in sent by itself back to the source (EIS). In this manner, changes for many assets deployed over many client/targets can be updated at the source/EIS by processing the single coalesced synchronization asset. This can decrease throughput requirements to the source (increasing speed) and decrease processing at the EIS.

Figure 21 is a flow chart of the EIS discovery asset adapter method/process (DAM) 2100. The DAM defines the process of discovering the assets 240 that need to be combined into a package 1100, also referred to as an execution unit. An execution unit has an execution graph (expressed or implied) associated with it, describing the path of execution as one process calls another. The DAM identifies each asset type, then identifies dependent assets that are candidates for inclusion in the same package.

In step 2110, the EIS calls the DAM to start the asset determination process. In a preferred embodiment, the DAM begins with the top-level page, typically a Java Server Page (JSP). The JSP is a server side script that is executed to create a web page. The choice of the web-page as the starting point for determining the dependency graph for the application is based on the utility of such a starting point. In other situations it may be appropriate to use a different starting point for determining the execution graph or dependency tree. Typically, the JSP is an XML based document that has both HTML and Java source code mixed together.

In step 2115, the DAM attempts to generate other asset candidates by identifying the HTML hyperlinks 2125b. Identification of these hyperlinks is based on a text search 2115a for image or navigation links in the JSP document. Once these elements are identified, their name is added to the asset candidate list. After identifying each candidate from the HTML, the DAM proceeds to step 2120. In other embodiments, the "edges" of the call graph are determined. This can be done by doing a text search for method calls in the program. See below.

In step 2120, the DAM identifies the java objects that the JSP will instantiate. The DAM generates the servlet class corresponding to the JSP. The class is generated by a well-defined mapping in the Java Servlet specification. The class is then compiled into Java byte codes. The byte codes are then examined to determine which classes are loaded by the servlet. In this way, the DAM determines which Java classes are used by the JSP. Once these classes are identified, the DAM includes those classes as asset candidates and proceeds to step 2125.

In step 2125, the DAM retrieves the next asset candidate off of the asset candidate list (ACL). The DAM then proceeds to step 2130. If the ACL is empty, the DAM proceeds to step 2180. Note that the ACL can be a data structure that is identical to the data structure of the asset definition data structure 1175.

In step 2130, the DAM determines if the asset is static HTML. This determination is based on the file extension being either "HTML" or "HTM". If the file extension is a match, the DAM proceeds to step 2135, otherwise, the DAM proceeds to step 2140.

In step 2135, the DAM searches through the HTML text looking for either "IMG" or "A" tags. These tags represent portions of the HTML document that would cause a Web Browser to initiate a load of an image or another HTML document. For each of the tags that match, the DAM creates another entry in the ACL. When the HTML document has been completely searched, the DAM proceeds to step 2125.

In step 2140, the DAM determines if the asset is a java class file (JCF). If the asset is not identified as a JCF, the DAM proceeds to step 2155. Otherwise, the DAM proceeds to step 2145.

In step 2145, the DAM generates an external call graph (ECG) for the JCF. The ECG is a list of the method calls that are made within the class represented by the JCF. The ECG is generated using decompilation technology. Decompilation technology is the capability of taking a compiled program and recreating the original source code. After the class has been decompiled, those methods made against external classes are placed in the ECG. The DAM then proceeds to step 2150.

Note that both in steps 2120 and 2145, the method calls are determined by either byte code examination or decompilation. Either method can be used in either step.

Decompilation is the inverse of compilation, translating an executable file into a higher level language. Examples of decompilation technology would include the following products: Mocha, DejaVu, WingDis, Jad, Jasmine, SourceAgain, GuavaD, and HomeBrew.

In step 2150, the DAM transforms the ECG into asset candidates. The DAM iterates through the ECG adding an entry in the ACL for each call that results in a unique asset descriptor. The DAM then proceeds to step 2125.

In step 2155, the DAM determines if the asset is a JAR file. If the extension of the file is not "JAR", "WAR", or "EAR", the DAM proceeds to step 2125. Otherwise the DAM proceeds to step 2165.

In step 2165, the DAM generates an external call graph (ECG) for the for each java class in the JAR file. The ECG is a list of the method calls that are made within the classes contained in the JAR file. The ECG is generated using de-compilation technology (see 2150).

In step 2170, the DAM determines if the assets from 2165 are EJBs. The DAM matches the method signature against that of the interface for EJB's. If there is a match, the DAM then proceeds to step 2175. Otherwise, the DAM adds the assets to the ACL and proceeds to step 2125.

In step 2175, the DAM determines the assets' EJB type. In a preferred embodiment, this is done by examining the descriptor entry for the bean. The DAM reads the XML deployment descriptor in the JAR file that contained the EJB. Searching through the XML deployment descriptor, the DAM finds the entry matching the filename of the EJB. There is an entry that specifies the EJB type. This type determines the asset type for the EJB. The DAM then proceeds to step 2180.

In step 2180, the DAM determines the data assets that are required for the assets that were identified. A search through the de-complied code for common database access methods yields table names, operations, and "where clauses". These are all common elements of database operations. These elements are each identified as an entity data asset type. It is expected that the database assets will need to be altered by a deployment engineer. If there is a "where" clause, the asset is an entity bean and the process 2100 proceeds to step 2190 where the type of the asset is set to "entity bean" (EB). However, if there is no "where" clause, the asset is a session bean and the process 2100 proceeds to step 2195 where the type of the asset is set to "session bean" (SB).

In a preferred embodiment, property files are searched for SQL queries. It is a common practice to store resource strings in property files. The strings associated with database statements would be likely candidates for being stored as resources.

In a preferred embodiment, additional assets would be discovered through the tracing of JSP chaining. JSP documents have the ability to specify other JSP documents that will be called from the current document.

In an alternative preferred embodiment, the byte codes for the Java classes would be disassembled and examined for SQL statements and external method calls. The disassembly of the Java byte codes contains a string pool. The string pool contains all the static strings that are declared in the class file. These strings provide candidates for SQL statements. The external calls in the disassembly are identified by the "invoke' mnemonic. Such calls that cannot be resolved to the current class are asset candidates.

In a preferred embodiment, the log of the web server is examined to determine which URLs are accessed. These URLs provide candidates for the start of call graphs for different packages.

In a preferred embodiment, the SQL statements are examined to determine tokens for targeting. The 'where' clause may contain column names that are accompanied by some binary operator with a literal string as the other operand. The configuration entry that specifies the Entity Data (ED) and/or Reference Data (RD) 'where' clause can be assembled from the 'where' clause by substituting the string literals with tokens. These tokens will later be replaced by the target asset adapter 1900. Existence of string literals indicates that the query should be part of an ED asset rather than a Reference Data asset.

In a preferred embodiment, the DAM determines if there are secondary tables that should be asset candidates. If the DAM determines that a table is a candidate for an ED asset, the DAM will examine the schema for the table referenced. From the schema information, foreign key (FK) relationships are determined. These relationships indicate secondary tables that are dependent on the current table, and show the relationship as indicated by the columns specified in the FK. If the Primary Key (PK) in the secondary table contains columns that match those columns that had tokens assigned during the previous embodiment, those tables become candidates for being Reference Data assets. Otherwise, any table that does not meet the previous condition is specified as an ED asset candidate. In a preferred embodiment, the union of the columns specified in the where clauses for a particular table specify those columns that need to be selected and created in the target environment.

This invention is not limited to web based applications. In alternative embodiments, other types of programs, e.g. C++, ASP, Visual Basic, Delphi, Fortran, etc. can be processed by the discovery method 2100 by generally searching through the program in the Execution Call Graph defined sequence and identifying all calls to external programs that need to be listed on the ACL. In alternative embodiments, the user can provide their own DAM.

The adjustment asset adaptor method 2100A, in Figure 21A, is used to determine on optimal way to process an asset, e.g. if an asset needs to be routed differently or executed on a different environment. When the adjustment method is called, it is given a context to examine. The context allows the adjustment method to determine specific information about the environment in which the adjustment asset adaptor method is running within. The context contains topics and/or constraints such as: routing, network load, CPU load, local license server location and/or availability, local certificate server, costing, and other topics that might allow the adjustment asset adapter to make decisions about how its assets 240 should be distributed in a more optimal manner. If the topic/constraint does not exist, the adjustment asset adapter method may add a query entry that will prompt various agents for the type of information that the adjustment asset adapter method needs in order to adjust the distribution of assets.

Some topics/constraints include load balancing, optimal distribution pathway on the network, closest ISP or local server, service level agreements, and migration.

One constraint that can be handled by the adjustment asset adaptor is load balancing. In one preferred embodiment, statistics based on the computational environment are fed back into the CDS/ADS. Theses statistics are then used to determine which computational environments are most available for executing program functionality. A common algorithm would be a round robin.

When providing asset distribution, it is useful to differentiate between the possible different routes an asset can take from source to target, and even the source from which the asset originates. The ability to track network load and determine optimal routing on the network can support Service Level Agreements in that a better load profile and routing path can be sold at a premium.

Migration occurs when assets that are deployed in one physical area of a network move to cause local areas of over or under use ("growth and shrinkage.") As an application is executed, or as a target moves through a network topology (e.g. a wireless device or other pervasive device physically moving with respect to the network), the actual asset distribution across the tiers of the network could change. If there is a greater demand for an application, the distribution system should move the appropriate assets toward the area of demand in order to reduce communication distances and to increase reliability in that smaller portions of the network will be used. However, the distribution of these assets might need to conform to constraints such as load balancing and service level agreements. Furthermore, as the demand lessens in parts of the network, it might be optimal to move assets from locations of heavy activity to those locations with less activity in order to free up resources for applications that have a higher demand. The same "growth and shrinkage" also can occur when a location of a target environment moves through a network topology, e.g. from a large city to a rural area. In this case, optimization can only occur if the target environment is tracked in a dynamic fashion and the assets can be physically moved to meet the changing criteria as enabled by the processes in this disclosure. In a preferred embodiment, the distribution of the assets is reconfigured over the network topology based on the demand caused by the execution of the application and/or or other criteria.

Another significant use of the Adjustment Asset Adapter Method can be to determine from where and through which route which asset is obtained or is executed if an application execution requires assets that are not in the local server. This function may need to invoke other adapter methods to query the available options regarding to the source, route and the choice of either executing remotely through bridging or transporting the asset into local server in order to execute its own logic to optimally select the asset arrangement for the application. This function along with other functions makes each of the nodes into a truly distributed server.

Figure 21A is a flow chart of the EIS adjustment asset adapter method/process (AAM) 2100A. The AAM defines the process of adjusting and measuring the distribution of assets. Adjusting the distribution is achieved by changing the package specification 1100. In a preferred embodiment, measurement uses the synchronization 2000 facility to pass metrics back

through the system, e.g. back to the source (e.g., EIS). The adjustment system can enable or disable a certain set of functionality to optimize the system based on external factors/criteria.

In step 2110A, various elements produce metrics, which are fed into the adjustment method's models (see below). (Elements are inputs to the performance models that can be provided by a user, designer, and/or some executing process, e.g. an adaptor method.) Performance metrics include, but are not limited to: data transfer between agents, transaction per second for computational environments, condition of a secure network connection, the number of clients per second that have been accessing a distribution server, or simply distance of assets from their target.

In step 2120A, different models that define some network optimization are run. For example, to optimize load balancing, a load-balancing model determines which servers have the capability to facilitate distribution of assets. To optimize quality of service, a QoS model determines the security and performance of the network paths for asset distribution. To optimize asset routing, a routing optimization model calculates the best path for asset distribution. Many of these models are well known in the prior art.

In step 2130A, the model produces results indicating the changes that need to be made to optimize the criteria the model is designed to optimize. In many preferred embodiments, this optimization requires the re-location, movement, and/or re-routing of assets and/or computational requests. These changes may result in an asset being moved to another cache, closer to a target environment or on a more secure network path. Other possible changes might include the choice of an alternate target environment for asset delivery, or the selection of an alternate computational environment in which to fulfill a request for execution.

These changes are applied to the distribution of the assets. In other words, step 2130A determines where assets should be in order to produce the optimization of the model.

For example, with a load-balancing model, each node can keep track of how many requests it can generate and how long each request takes. There would be some adaptive grouping of different requestors into groups for load balancing, the request statistics would determine which intermediate computational environment would service a request. For distribution, we would keep track of the CDS/ADSs that were providing the assets to each computational environment. If one CDS/ADS was providing an inordinate number of assets, that CDS/ADS might refer the requesting CDA to a peer CDS/ADS.

In another example, with the Service level agreement model, a service token is added to all transfers. When a package needs to be transferred or processed, the service token is examined. If there are other packages that have a higher service token, those packages are transferred or processed before the lower level tokens or process on an environment with better resources.

In another example, the network routing optimization would have a goal of asset migration. Request statistics are kept for each asset based on the number of times the asset was

distributed and how many times the deployed asset was accessed in the client environment. Those assets that are deployed frequently in a target environment would be distributed physically closer to the client and perhaps even deployed at the client/target. As the target demand for a particular asset decreased, the physical location of that asset could be moved further from the target.

In step 2140A the process determines where the assets to be moved are, and re-directs the distribution of these assets to the location determined in step 2130A. In a preferred embodiment, step 2140A re-directs these assets by changing the node ID 1524 in the node registration specification 1522. Note that the node ID can refer to a CDS/ADS, EIS, proxy server, or any other computer on the network in addition to any target. In some embodiments, one or more of the assets is directed through a route that meets certain routing criteria, e.g. routing under a certain speed threshold or through certain nodes. This route determination and criteria are well known.

Now that the adapters of the invention have be disclosed and explained, a more detailed explanation with be given about the asset lifecycle 240L described in Figure 2A. The asset goes through a lifecycle starting in the source tier, moving through the deployment tier, into the target tier, and then optionally back through the deployment tier to the source tier and/or to any other node or nodes in the network if required. The asset adapter methods define the steps in this lifecycle where asset type specific processing is required for the asset to continue through the lifecycle.

In the source tier, resources are discovered using the discovery asset adapter method 2100, to identify candidates for classification as assets and, through the export adapter method, to package assets together as packages. In a preferred embodiment, a package specification 1100 is created that in turn contains asset specifications 1170. The asset specification 1170 is stored in the deployment tier until a package 1100 is scheduled for delivery.

The version asset adapter method 1660 is used to determine the current version information of the assets 240 in the source tier. This version information is compared with the target tier asset version information in the deployment tier to determine if assets need to be deployed from the source tier to the target tier.

The export asset adapter method 1600 is used to obtain the actual current version of assets in the source tier that need to be distributed to the target tier. After the assets are exported, the assets are preferably moved to the deployment tier and stored in an asset cache. When exporting assets, the export asset adapter method captures the logic, data, and extended environment information 220 for an asset 240 and puts it into an asset data structure 240.

If an asset requires processing the processing may be done when the asset is stored in the asset cache or at any time before the asset is distributed to either a secondary cache in the deployment tier or the target tier. The processing is primarily performed on the asset's extended environment 220, in an attempt to translate the extended environment 220 to run in harmony with the base environment 250 in the given target tier. However, the processing

process may also change the logic/data portion of the asset or both the logic/data and the extended environment portion of the asset.

An agent in the target environment requests the assets that are pending for delivery to that target tier. The target processing asset adapter method 1900 is executed against any asset that requires targeted processing before being sent to the target tier. Target processing is intended primarily to change the Logic/Data section 210 of the asset data structure 240 in order to provide a unique asset that can create or has personalized information for the specific target in which it is being deployed. The targeting can be for an intermediate target (a server that will in turn serve many users) or a final target (a single node that will serve a single user).

When the asset is sent to the target tier, the deploy asset adapter method 1700 is invoked to deploy the asset 240 into the computational environment, i.e., the base environment 250, in the target tier. The extended environment 220 from the asset's data structure 240 is used to set the base environment 250 and extended environment 220 in the target tier to run the asset in a correct manner. The asset's logic and data 210 are then deployed into the base environment 250, and since the base environment 250 has been adjusted, the logic 210 will function correctly and the data 210 will be accessible.

When changes happen in the target tier that warrant synchronization, the synchronization asset adapter method 2000 is executed to create a synchronization asset and the asset 240 is propagated back through the deployment tier into the source tier and the source tier resource that corresponds to the synchronization asset is synchronized with that asset.

Agent methods are now described in detail. An agent is a process or service that can be assumed to be available in an environment. For example, the EDA is available in the source environment, the CDS/ADS is available in the distribution environment, and the CDA is available in the target environment. When the EDA, CDS/ADS, and CDA agents are described, it is assumed that we are referring to a simple network topology with three distinct tiers: source, distribution, and target. If we consider an arbitrary network topology, with overlapping combinations of these tiers, then it is advantageous to have more abstract agents to describe the functionality. Some of these preferred, more abstract agents are: Publish, Subscribe, Computational, and Caching.

These agents provide a generalized framework for distributing assets 240 over the network. The types of assets they distribute correspond with the asset adapters that are defined for the system. For much of this disclosure we refer to a set of asset adapters specific to distributing Sun's J2EE web and enterprise applications as a non-limiting example. However, other assets, as described above, can also be distributed.

In a preferred embodiment, the common agents in the simple distribution topology are typically comprised of several agents, some common aggregations are described here. The EDA is comprised of a publishing agent. The CDS/ADS is comprised of a subscriber (subscription) and caching agent. The CDA is comprised of a subscription and computational agent.

Each agent is responsible for providing a service interface, on a network node, that other agents can access. Those other agents may be on the same node/computer (to distribute and execute assets within tiers or subdivisions of the computer) or on different nodes/computers on the network.

Figure 21B is a flow chart of an alternative preferred discovery asset adapter process 2100B.

The discovery asset adaptor process (discovery process) 2100B identifies member objects of one or more computer system parts 100F and establishes at least one topographical relationship among the member objects.

The discovery process 2100B starts 2105B by identifying all of the member objects.

This identification can be done manually 2130B, e.g., by a systems engineer examines the system parts 100F typically by traversing the directories of the system parts 100F, and manually produces an Asset Inventory 2150B and an Intermediate Representation 2160B.

Alternatively, in an automatic embodiment of the discovery process 2100B, a process 2140B traverses one or more computer file systems to find one or more of candidate objects of the system part 100F that are located on the computer file system. Examples of less sophisticated processes that traverse one or more computer file systems include the DIR function in Microsoft DOS, the Microsoft Windows Explorer, and the LS function in the UNIX operating system. These functions are well known but only name the objects found within the topographical scope of the parameters that are provided to these functions. These functions do not recognize or attempt to recognize any relationship among these objects other than the topographical relationship of their location. These functions are not designed to identify any objects found as being candidate member objects of any system part 100F. Therefore, these functions cannot guarantee that all of the members of any system part will be identified. Therefore, the objects identified by these functions cannot be considered candidate member objects of a subject system part 100F.

The automated identification step 2140B overcomes this difficulty by using knowledge about the specific system part 100F and the EIS system environment in an expert system designed for this purpose. For example, for an expert system to identify the candidate member objects of a common implementation of a MS-Win32 software application, the expert system, given the name of the system part 100F (for example, "Payroll" for a payroll application 108) and a root starting location in the EIS computer file system, would traverse the computer file system starting at the root location, would search for a branch of the computer file system that matched the branch of the computer file system named in the expert system's rulebase as the system part's "home" directory. The expert system would have a high expectation (expressed internally as probabilities) that objects found in the system part's "home" directory were, in fact, candidate member objects of the system part, and would add these objects to the Asset Inventory 2100D and to the intermediate representation 2100C (see below) of the system part.

Next, the expert system would traverse the computer file system to the well known common application directories of an MS-Win32 operating system. There, the expert system would search for runnable library files (such as ".lib") or runnable runtime files (such as .dll) named in the expert system's rulebase specifically as the common application files required for correct execution of the system part (i.e. "payroll"). Next, the expert system would traverse the computer file system to locate the operating system specific configuration file directories (for example, in MS-WINNT 4.0, the "WINNT" directory), and there, would search for those configuration files and files which contained configuration entries (such as the WINNT Registry) and add those files to the Asset Inventory 2100D and to the intermediate representation 2100C. Only after all directories and computer system files named or suggested in the expert system's rulebase were examined, and all candidate member objects had been placed in the Asset Inventory 2100D and intermediate representation 2100C would the expert system report completion of its task, which is production of the Asset Inventory 2150B and production of the Intermediate Representation 2160B. Any search objectives or file acquisition requirements stated in the expert system's rulebase that the above described process failed to locate and add to the Asset Inventory 2100D and the intermediate representation 2100C would be reported to a human deployment engineer, who would finalize the production of the Asset Inventory 2150B and Intermediate Representation 2160B. Any sufficient level search result would be acceptable. In one preferred embodiment, sufficiency is based on the goal of being able to later distribute and execute the assets in a base environment on one or more target nodes.

When the Asset Inventory 2100D and the Intermediate Representation 2100C are complete, the member objects so inventoried and identified are considered digital assets 240 members of the system part 100F. These digital assets 240 form the parts of a part-whole relationship with the system part 100F.

Alternatively, the human deployment engineer could develop additional rules to place in the expert system's rulebase so that future invocations of the expert system would succeed in locating all files and completing all searches called for in the rulebase. Although the use of an expert system for discovery of assets is believed to be new, expert system technology is well known in the art.

Now more detail will be presented about the steps that the expert system (or the human deployment engineer) had to perform in the above described discovery process 2100B.

The Intermediate Representation 2100C is created by placing an identifier as a node in the Intermediate Representation for each member object discovered. Before the member objects are recognized as digital assets, these identifiers are member object identifiers. Subsequent to the recognition of the member objects as digital assets 240, the member object identifiers become or are transformed into digital asset identifiers. In a preferred embodiment, the intermediate representation 2100C is a graph with nodes and edges. The digital asset identifier occupies one of the nodes of the graph. The edges represent the topographical relationship of the digital assets with respect to each other and with respect to the EIS computer file system.

The digital asset is created from the member object, chosen as described above, by placing the member object in a logic/data section 210 of the digital asset 240 and by further creating an extended environment data structure 220 (described above) that is attached to the logic/data section 210.

As stated above, the digital asset is placed in an asset inventory container object called the Asset Inventory 2100D.

In some embodiments, a definition of each of the digital assets is optionally entered or completed in an asset definition data structure 1170, above. As already stated, these entries have descriptions of one or more digital asset attributes of the digital asset 240.

The member objects/digital assets are processed, as above, until the Intermediate Representation 2100C fully describes the system part 100F and the asset inventory container object (Asset Inventory) 2100D has a complete inventory of the digital assets of interest from the system part 100F. The asset definition data structure 1170 is then also a complete list of the digital assets 240 of interest in the system part 100F.

The discovery process 2100B further stores one or more descriptors in the extended environment 220 of each digital asset after the extended environment of the digital asset is created.

In one embodiment, the descriptors include one or more common descriptors 210B that provide a unique identification of the digit asset on the network(s). These common descriptors 210B can include one or more of the following: a digital asset name of the digital asset, a unique fully qualified name of the digital asset, an address of the digital asset, and a size of the digital asset. These common descriptors are obtained from the EIS at the time the digital assets are discovered.

In one preferred embodiment, the descriptors include one or more asset dependency descriptors 220B. The asset dependency descriptors include any one or more of the following: one or more names of other digital assets on which the digital asset is dependent 222B, an asset identifier, and one or more unique fully qualified names of other digital assets on which the digital asset is dependent 222B. These fully qualified names are also obtained from the EIS.

In one preferred embodiment, the descriptors include one or more reference descriptors 260B. The reference descriptors 260B include any one or more of the following: a reference link descriptor, a reference file descriptor, and a reference directory descriptor. The reference link descriptor may provide a world-wide-web (WWW) address that has contents used for processing of the digital asset. The reference link descriptor may also provide a world-wide-web (WWW) address that has contents used during execution of the digital asset. The reference directory descriptor provides additional address information that is used to locate

one or more of the associated digital assets. The reference file descriptor is a unique fully qualified name of a file required for reference by the digital asset.

In one preferred embodiment, the descriptors include one or more runnable descriptors 240B that indicates that the digit asset 240 is a runnable digital asset. The runnable descriptors 240B may include a description of the EIS execution environment. For example, the runnable descriptor might be Window NT 4.0, Linux version 5.0, or Solaris version 6.1.

In one preferred embodiment, the descriptors include one or more non-runnable descriptors 242B that indicates that the digit asset is a non-runnable digital asset. The non-runnable descriptors 242B may include a description of the EIS execution environment as above.

Figure 21C is a diagram of a prior art graph structure, specifically a containment graph structure, used to establish membership relationships of digital assets.

In the preferred embodiment, the graph structure can be used as the Intermediate Representation. For Figure 21C, in order to provide additional detail for explanation purposes here, the nodes of Figure 21C have been drawn to illustrate visually the different digital asset types that are the subjects of the discovery process. Further, text identifications of the digital asset type have been placed in a number of the illustrated nodes for further clarity. In practice, a graph is represented in computer memory as a data structure, containing both node data structures and edge data structures. See well known works by Prim, Kruskal, Dijkstra, et. al. which are herein incorporated by reference in their entirety. Well known algorithms dictate correct methods for traversal of graphs of this type. Further, there exist well known methods described to construct such a graph given the topological relationships discovered or examined by the discovery process as it follows the rules contained in the expert system rulebase or traverses the EIS computer file system. In Figure 21C, as non-limiting examples of the diversity of digital assets which may be discovered, digital assets such as Runnable Script File 2130C, Non-Runnable Document 2140C, Runnable Java Class 2130C, Runnable J2EE EJB File 2132C, Non-Runnable XML File 2135C, Non-Runnable HTML Document 2145C, Runnable .EXE file 2130C, Non-Runnable Data File 2150C are identified. Further, the topological relationship of these digital asset types is demonstrated for non-limiting example purposes by their respective locations in branch directories of the example computer file system, such branch directories being identified as "Users" 2112C, "Docs" 2110C, "Statistics" 2120C, etc.

Figure 21D is a block diagram of a preferred asset inventory 2100D. This data structure contains the assets identified by the discovery process 2100. In a preferred embodiment, the assets 240 that occupy each entry, include the logic/data sections 210, EE sections 220, and optionally the asset interface 230 of the discovered digital assets 240.

Figure 22 is a flow chart of a publishing agent method 2200. The publishing agent method (PAM) handles requests for the discovery 2100, versioning (see Figure 16A), and export 1600 of assets. The PAM groups together these lifecycle steps (i.e. adapter methods) in the asset lifecycle into a single agent.

Discovery 2100 is the process of determining asset dependencies that result in the definition of a package specification 1100 by means of the export adapter method 1600. Versioning (see Figure 16A) is the process of determining the current version of an asset against a cached asset and updating the cached asset if it is out of date. Export 1600 is the process of taking the various elements that constitute an asset and packaging those elements into a data structure.

In step 2205, the PAM receives a request to discover the assets that are used in one or more applications. An initial asset is specified to begin the discovery process, then the dependent assets are discovered until the PAM reaches a terminating condition.

In step 2210, the PAM receives a request for the current version information for specific assets. The PAM looks to assets in the local environment to determine the version information.

In one embodiment, the PAM uses the time stamp associated with the asset as the version information.

In step 2215, the PAM receives a request for a data structure containing assets. A list of assets accompanies the request. The PAM exports the current version assets from the local environment into a data structure. The PAM returns the data structure to the caller.

In a preferred embodiment, the PAM splits the data structure into smaller pieces that may be of a more appropriate size for sending over a network.

Note that the PAM may run steps 2205, 2210, and 2215 in any order depending on how the requests of the PAM are received. In addition, the PAM may omit one or more of these steps. Essentially, the PAM is a convenient way of grouping these asset life cycle steps that typically occur in the source tier. Therefore, a set of asset requests can be sent to the PAM and the PAM, a single agent, will perform the necessary discovery, versioning, and exporting that is normally required at the source.

Note that the PAM can be considered as a package 1100 of assets 240 distributable, and with a life cycle, of its own. By distributing the PAM package of assets to any given source, that source will be enabled to discover, version, and/or export assets as required.

Figure 23 is a flow chart of a subscriber agent (SA) method 2300. The subscriber agent method (SAM) 2300 is responsible for an adjustment asset lifecycle step, supporting bridged computational environments (proxy capabilities, below), and dispatching requests between other agents.

The SA supports the adjustment asset lifecycle step by logging and reporting the interaction between all of the agents that contact the SA.

The SA can provide the proxy capabilities described in the computational bridging adapter method. When requests are sent to the SA, those requests are routed to the source computational environment. See computational agent below. When responses to a proxy request are returned from the source computational environment, the response is sent to the target computational environment, a computational agent running on the target.

The SA also can coordinate the transfer of assets and information between the publishing agent and caching agent. The SA performs this functionality for discovery, versioning, and exporting.

The SA also can provide the transfer of assets and information between the source and target environments for synchronization.

The SA also can coordinate the transfer of assets information between the caching agent and computational agent. The SA performs this functionality for deployment and synchronization.

In step 2310, the SA is instructed, by the CDS/ADS, to request that the asset discovery be executed by the source publisher agent 2200. The package specification 1100 from the export process 1600 is then received and stored in the distribution tier.

In step 2315, the SA is instructed, by the CDS/ADS, to request that the asset versioning method (Figure 16A) be executed by the source publisher agent 2200. The resulting version information from the versioning process is stored in the distribution tier.

In step 2320, the SA is instructed, by the CDS/ADS, to request that the asset exporting method 1600 be executed by the source publisher agent 2200.

In step 2325, the assets resulting from the exporting process 1600 in step 2320 are sent to the caching agent 2500 on the same node as the SA so that they can be stored in one or more of the asset caches.

In step 2330, an SA, typically in the target environment, requests 1400 a manifest 1452 detailing the assets to be distributed to the target environment from the SA in the distribution tier.

In step 2335, the SA in the distribution tier requests the manifest 1452 detailing the assets to be distributed to the target environment from the distribution cache and returns it to the requesting SA in the target environment.

In step 2340, the SA, typically in the target environment, is optionally requested to synchronize 2000 an asset. The SA proceeds to conspire with the SA in the distribution tier to forward the synchronization asset.

In step 2345, the SA in the distribution tier contacts the computational agent 2400 in the source environment and sends the synchronization asset to that agent for synchronization.

Note that the SA can be considered as a package 1100 of assets 240 distributable, and with a life cycle, of its own. By distributing the SA package of assets to any given source, CDS/ADS, general server, and/or client/node will be enabled to communicate with and process requests from other SA's, and coordinate the transfer of assets and information, as required.

Figure 24 is a flow chart of a computational agent method 2400. The computational agent method (CA) is responsible for providing the functionality involved with deployment of assets into a computational environment, and generating synchronization assets for other environments.

Additionally, the CA enables the installation of the base environment 250 as described in Figure 26 below.

The computational agent 2400 is also responsible for providing the computational environment for assets. The stewardship of the environment involves providing the services that the assets need, either directly or by proxy. In the case that a service is needed but not available, the CA initiates the distribution of the asset 240 with the appropriate extended environment 220 that is needed.

In step 2410, the CA 2400, typically in the target environment, is requested to deploy 1700 an asset 240 or set of assets. The CA directs the request to the deployment asset adapter 1700, which performs the actual processing.

In step 2420, the CA 2400 in the source environment is optionally requested to synchronize 2000 an asset that the subscription agent 2300 forwards from the target environment. The CA selects the proper synchronization asset adapter 2000 to complete the synchronization. Typical this sychronization is done in a way specific to each asset type 240T.

In step 2430, the CA 2400 in either the source environment or distribution tier either fulfills or forwards a request for execution. If the CA has the asset 240 that can fulfill the request, the CA runs the request against the asset, returning the result to the caller, which may be the target environment or a subscription agent 2300 that is forwarding a request. If the CA does not have an asset to fulfill the request, it calls the subscription agent, typically on the same node, in order to forward the request to another CA. In the case that requests reach the CA in the source environment, the requests are made directly against the resources in that environment.

Note that the CA can be considered as a package 1100 of assets 240 distributable, and with a life cycle 240L, of its own. By distributing the CA package of assets to any given source, CDS/ADS and/or general server will be enabled to process, deploy, synchronize, and cache assts, as required.

Figure 25 is a flow chart of a caching agent method 2500. The caching agent method CAM) 2500 is responsible for the processing and targeted processing of assets. These functions allow the cached assets to be processed for distribution.

The processing functionality is a general transformation that would be performed on an asset, possibly for normalization. Normalization would allow the asset to be stored in a neutral format, being of some advantage to the distribution process. Neutral formats typical involve changes to the extended environment 220 to make the extended environment 220 less specific than the source environment and enabling the extended environment to execute more easily with a variety of base environments 250 in the target environments. An example or a Session Bean (SB) or entity bean (EB) where the source format could be a first application server and the target format could be a second application server, a neutral format would be one specified by, e.g., strictly adhering to, the J2EE specification.

The targeted processing function recognizes the need to transform the asset for the specific target environment for which it is targeted. This recognition is a result of the package specification 1100 indicating that a give asset should be targeted at a specific target. This information is given in the other field 1179A in the respective asset definition data structure 1170. The information need for each of the specific target is given is a specification (node specification). In a preferred embodiment, this is a list of name/value pairs with a token label accompanied by a substitution value for each pair.

The CAM also can be responsible for creating and maintaining an asset cache.

In step 2505, the CAM is requested to perform targeted processing (the processing need for the asset to execute on a specific target) an asset or set of assets. The CAM directs the request to the processing asset adapter 1800, which performs the actual targeted processing. In a preferred embodiment, this targeted processing occurs after the asset is cashed to reduce overhead of storing a targeted asset for each different type of target. However, targeted processing could be performed before caching.

In step 2510, the CAM is requested to perform processing 1800 on one or more assets. The CAM forwards the request to the processing asset adapter 1800, which performs the processing.

In step 2520, the CAM is requested to perform targeted processing on one or more assets. The CAM forwards the request to the targeted processing asset adapter 1900, (in a preferred embodiment residing on the distribution server), which performs the targeted processing specific for the respective target.

In step 2530, the CAM is requested to store an asset in the asset cache. In a preferred embodiment, the CAM performs the functionality that is required to store the asset, and manage the cache, including: deleting old assets, updating database tables, and notifying the proper subsystems that a change has occurred in the asset cache.

In step 2540, the CAM is requested, via a manifest, to read assets from the asset cache and return those assets to the caller.

Figure 26 is a flow chart of an asset distribution process. In a preferred embodiment, the CDS/ADS stores assets corresponding to the following asset classes: applications, asset adapters and agents (see below) and base environment. While these three classes of assets enable different functionality, they are each seen by the CDS/ADS as being assets 240 (or packages 1100 of assets 240) as described in this document. The difference between these asset classes centers around the context each of these classes is intended to support.

The assets associated with the application comprise the actual application elements or assets 240 that are distributed by the CDS/ADS. The application assets are "first class" assets in the entire system. Typically, these assets (assets used in applications and/or sub applications) are the major workload for the distribution server, CDS/ADS, and agents, e.g. EDA and CDA.

In addition, the asset adapters and agents (e.g. asset packaging 1300, client deployment process 1400, node registration 1500, versioning 1660, export 1600, client deployment 1700, process 1800, target 1900, synchronize 2000, discover 2100, adjustment 2100, publishing 2200, subscriber 2300, computational 2400, caching 2500, system asset distribution 2600, streaming 2700, bridging 2800, QoS 2900, and any other process that distributes and/or changes assets) can be themselves assets 240, and can be distributed and processed to provide the necessary asset lifecycle functionality in the source tier, distribution tier, and target tier. In a preferred embodiment, the EDA, CDS/ADS, and CDA are intelligent enough to determine the type of asset for each step in the asset lifecycle, but they defer the actual asset operations to the individual asset adapter. The actual asset adapters are themselves applications, therefore, in some preferred embodiments, they have package 1100 and asset specifications 1175 defined that allow them to be distributed. In these cases, the asset adapters can be loaded on demand, as the EDA, CDS/ADS, and CDA need to process each individual asset.

The base environment represents those assets that are needed and are within which the other assets 240 run -- for example: a web server for static content assets (SC), a servlet engine for JSPs (JSP), a Java Runtime Environment for Java class assets (JB), an application server for the EJB assets (SB and EB), and a DBMS for the data assets (RD, ED, and EB). Each of the servers that make up the base environment themselves can be applications, preferably Java applications, allowing them to be defined in package 1100 and asset specifications 1175 for distribution. The Java runtime environment is a native application that is dependent on the operating system in the target environment, and is distributed using standard installation programs, which are widely available. The CDA is distributed by the boot-strap process described below.

There are dependencies between these different asset classes. There exists an implicit relationship between the asset types and the corresponding asset adapters, as described above. If an asset of a certain type is going to be distributed, the asset adapter for that asset type needs to be distributed before the dependent asset to the particular location (e.g. tier) of the network.

An example would be a static content asset type would need a static content asset adapter in order to perform any of the distribution functionality on the asset, for example at the CDS/ADS.

There exists an implicit relationship between the asset types 240T and the base environment 250. If an asset type is going to be deployed, it requires a certain runtime environment. The base environment assets are distributed in order to provide this environment. For example, a JSP asset would require the assets that comprise the web server and servlet engine to be installed in the target computational environment before the asset could be expected to run the asset in that environment. In some embodiments, these base environments 250 are pre-installed. In other embodiments, the base environments 250 are distributed as one or more base environment 250 assets 240 prior to their being used. The base environment 250 assets 250 could also be distributed to required locations on the network by alternate, well known network distribution techniques.

In some embodiments, one or more asset adapters and/or agents are considered part of the base environment 250. For example, in the CDS/ADS system process adapters, target adapters, and/or subscriber agents can be part of the base environment 250 and could be distributed or pre-installed at the CDS/ADS.

In step 2605, the distribution agent (DA), more specifically the EDA (at the source), CDS/ADS (at the distribution level), or CDA (at the target/client), determines which asset adapters are needed for the asset types that have been discovered. The asset types are specified in the asset specification 1175.

In step 2610, the DA determines if those asset adapters are in the local environment, e.g. the target/node. The local environment contains a predefined configuration data structure indicating the asset adapters that are in the local environment. In a preferred embodiment, the data structure would be a properties file. For the adapters that are not in the local environment the process proceeds to step 2615.

In step 2615, the DA requests the packages1100 associated with the asset adapters not located in the local environment for immediate distribution to the current node. These asset adapters come from the asset adapter asset cache, or, in a preferred embodiment, the system asset cache on the CDS/ADS. Note that assets that are not in the local environment might include later versions (see versioning 1660) of the respective asset 240.

Examples of asset adapters include: SC, JB, SB, EB, RD, & ED which correspond to the respective asset types on which these asset adaptor operate.

In step 2620, the DA determines which base applications are needed for the asset types associated with the assets that are defined to be used at a particular computer on the network. The asset types are specified in the asset specification 1175 located in the respective environment. For example, one or more client/nodes would subscribe to a particular service. The clients would have different operating systems/platforms. The service, along with the

client platform/environment, would determine the set of assets and the set of adapters/agents required. These assets 240 would be pre-installed and/or distributed to the client from the CDS/ADS (or other proxy servers). As a further example, the CDS/ADS would need all the adapters necessary to process, target, etc. all of the combined target assets that would need to be distributed.

In step 2625, the DA determines if those base environment 250 assets in the base environment are needed in the local environment. (In embodiments where the assets required in the base environment are fixed and known, this step 2625 can be omitted.) The local environment contains a predefined configuration data structure indicating the base environment Assets that are in the local environment. In a preferred embodiment, the data structure would be a properties file. For the base applications that are not in the local environment the process proceeds to step 2630.

In step 2630, the DA requests the packages 1100 associated with the base environment not located in the local environment for immediate distribution to the current node. These base environment assets come from the base environment asset cache (typically on the target), or, in a preferred embodiment, the system asset cache on the CDS/ADS.

Examples of base environment 250 include, but are not limited to: web server, servlet engine, EJB container, DBMS, and CDS/ADS.

In optional step 2640, the DA would continue on to perform whatever operations were in progress before the check of the asset adapters and base environment 250was performed. In general, the DA or agents like this, e.g. the EDA (source/EIS), the CDS/ADS (at any distribution server), and the CDA (client/target), perform these functions on any network system on any tier of the network so that the systems have the required adapters to process their assets and the required local base environments to deploy their assets. As described above, these assets can be dynamically changing as the move across the network tiers during their life cycle 240L.

In a preferred embodiment, the DA would be able to bootstrap its entire environment. One notable exception would be base environment 250 that have specialized installation requirements that cannot be supported by the asset adapters. An example of a non-distributable application would be a DBMS that changes operating system registry entries.

Asset streaming reduces the transmission of duplicate data. Frames are defined as complete assets. Deltas are the difference between two versions of an asset. One can incrementally apply the deltas against a prior frame to create a complete asset for the new version associated with the delta(s) of that asset. Therefore, the complete new version of the asset does not have to be distributed, but only the deltas that exist between the older (prior) version and the new version. Potentially, asset updating/streaming reduces the bandwidth for data transmission and offers opportunities for merging synchronization assets.

In some preferred embodiments, the distribution system performs optimizations to decrease the redistribution of assets to targets or distribution servers. For example when an asset has changed, often the difference between the old asset and the new asset is only a small difference. The streaming asset provides the additional optimization of identifying the parts of an asset 240 that have changed and, therefore, the parts that only need to be sent. By decreasing the amount of information that is sent, the distribution can utilize less bandwidth and increase the amount of network throughput for other applications. Streaming assets represents a major conceptual shift in the distribution of applications and data across tiers of a network.

For web-based applications, streaming can greatly reduce the amount of redundant data that is sent across the network. For example, an online brokerage application would want to supply up-to-date portfolio information and order status information to their customers. With existing technologies, the users would poll the server repeatedly to gain this information. The server would be overloaded at peak times when everyone would want to know his or her portfolio status. The online brokerage would need to increase the network and EIS resources to make their systems available under these circumstances, although this scenario is not representative of their normal user load. Asset based distribution would decrease the interaction with the servers at the EIS of the online brokerage. Only the changed data would be sent down to the user, the static content, dynamic content, and EJBs would be located on the user's PC, or local server. The amount of sent data is greatly reduced between the EIS and the user, and the EIS performs far less processing because it doesn't have to generate the web page. The streaming asset goes one step further, assuring that only the data that has changed is sent to the client. If only the pricing data has changed, that is all that is transferred to the user. If a small part of a single JSP has changed, only that small change is sent to the user.

By combining streaming assets with QoS 2900 capabilities (below), the CDS/ADS is able to support streaming media such as voice and video.

Figure 27 is a flow chart of a streaming process 2700. In a preferred embodiment, the steps of this process 2700 are an optimized supplement to steps and methods described elsewhere in this document.

In step 2705, the Export Asset Adapter (EAA) determines that the asset specification indicates that the current asset should be treated as a streaming asset. The EAA then performs a test 2707 based on the asset type to determine if a frame or delta needs to be generated. On an asset-by-asset basis, this is done to determine the differences between a prior asset, e.g. located on a client, and a current asset (latest version) typically located on the source/EIS. In a preferred embodiment, a copy of the prior asset is retained at the source/EIS to facilitate this test/comparison. If a frame needs to be generated, the EAA continues to step 2710. If a delta needs to be generated, the EAA continues to step 2715.

For example, if the asset type is an EB, ED or RD, the EAA would query the data and based on the amount of data that has changed, would either generate a frame or a delta.

In step 2710, the EAA continues to export the asset as it normally would in the export asset adapter method 1600. The frame asset can be considered the same as the asset described in the other methods, and treated in the same manner. The EAA indicates that the asset is a frame asset by setting a value in the asset data structure 1175 to indicate this state.

In step 2715, the EAA creates a delta for the asset. A previous copy of the asset is compared with the current asset. The difference between these two assets is the delta that will be used to create the delta asset. The resulting delta asset represents the changes that would need to be applied in the target environment that has had all the previous deltas applied to the last frame.

For example, if the asset type is EB, ED or RD, the EAA would query the data from the source database and would either generate the delta. In one embodiment, the EAA would generate a version of the last asset by applying the deltas against the last frame. The EAA would then perform a row-wise (record – wise) differentiation between the last asset and the current data. Then a (table) column-wise differentiation would be applied. (Difference or comparisons functions are well known.)

In step 2720, the EAA continues to export the asset as it normally would in the export asset adapter method 1600. The delta asset is treated the same as the asset described in the other methods, and treated in the same manner (e.g. sent to the CDS/ADS). The EAA indicates that the asset is a delta asset by setting a value in the asset data structure 1175 to indicate this state.

In step 2725, the CDS/ADS receives an asset and determines whether it is a streaming asset or a normal asset. If it is normal, the CDS/ADS continues with normal operations, i.e., process 2700 terminates. If the asset is a streaming asset, the CDS/ADS continues to step 2730.

In step 2730, the CDS/ADS determines whether the asset is a frame or delta. If the asset is a frame, the CDS/ADS goes to step 2735. If the asset is a delta, the CDS/ADS proceeds to step 2740.

In step 2735, the CDS/ADS finds deltas and last frame for the asset in the asset cache and marks them for removal. The CDS/ADS then caches the current frame asset and proceeds to step 2745.

In step 2740, the CDS/ADS caches the delta asset in the asset cache and proceeds to step 2745.

In step 2745, the CDS/ADS resumes normal operations as if the asset was not a streaming asset.

In step 2750, the CDS/ADS receives a request to process an asset. If the asset is a streaming asset, the CDS/ADS proceeds to step 2755. Otherwise the CDS/ADS proceeds to process the asset as described in the process asset method 1800.

In step 2755, the CDS/ADS selects the Processing Asset Adapter (PAA) based on asset type. The CDS/ADS requests that the PAA create the complete asset by applying the deltas to the frame asset in order of creation then proceeds to step 2760.

In step 2760, the PAA applies the differential algorithm for that asset type against the processed asset to produce a new delta. In a preferred embodiment, the PAA would create a frame asset if the combination of the new delta and the existing deltas for the asset are larger than the frame.

In step 2765, the CDS/ADS stores the new delta asset in the asset cache.

In step 2770, the CDS/ADS builds a manifest for a target/client that includes a streaming asset. If 2771 the client has the most recent frame asset, the method continues onto step 2772, otherwise the method goes to step 2774.

In step 2772, the CDS/ADS determines the last delta that the client has received, and adds the subsequent deltas for that asset to the manifest. The method continues to step 2780.

In step 2774, the CDS/ADS adds entries for the last frame and subsequent deltas for the specified asset.

In step 2780, if the EAA is deploying a streaming asset, it deploys the frame and/or deltas into the target environment. For different asset types, the application of the delta file has different meanings. Overall, the application of the delta is similar to replacing the existing asset.

In step 2782, if the asset is registered with synchronization capabilities, the EAA saves a copy of the asset for later comparison.

In step 2784, the CDA is requested to synchronize target assets with source assets. The CDA selects the appropriate Synchronization Asset Adapter (SAA) for the specific asset type. If the asset is classified as a streaming asset, the method proceeds to step 2786

In step 2786, the SAA creates a delta for the asset. A previous copy of the asset, created in step 2782, is compared with the current asset. The difference between these two assets is the delta that will be used to create the delta synchronization asset. The resulting delta synchronization asset represents the changes that would need to be applied in the source environment that has had all the previous deltas applied to the last frame that was sent.

In step 2788, the SAA continues to synchronize the asset as it normally would in the synchronize asset adapter method 2000. The delta asset is treated the same as the asset described in the other methods, and treated in the same manner (e.g. sent to the CDS/ADS). The SAA indicates that the asset is a delta asset by setting a value in the asset data structure 1175 to indicate this state.

In a preferred embodiment, the CDS/ADS might coalesce deltas from multiple targets before sending those deltas onto the source. This would reduce the processing in the source environment and also reduce network traffic.

In step 2790, the SAA applies the synchronization information (e.g. LD 210 and/or EE 220) in the synchronization asset 240 of the source environment. If the asset is a streaming asset, the frames and deltas are applied to the source environment.

Figure 28 presents the Bridged Computational Environment (BCE) capability of the DIS technology. BCE provides the runtime capability to maintain communication between runnable application parts. The communication can be maintained through a variety of mechanisms, including but not limited to: redirection, server proxy, object proxy, and API proxy. The BCE maintains efficient communication between runnable application parts s that have been distributed in was beyond the intention of their design.

In step 2805, a method within a deployed asset attempts to access a resource that will cause a fault. If the resource is in the J2EE API, the method proceeds to step 2810.

In step 2810, the J2EE API fault is handled. If 2812 the fault is in a resource that has been defined in the JDNI that should be accessed on a remote machine, the method proceeds to step 2815. Otherwise, the method proceeds to step 2820.

In step 2815, the JNDI implementation in the target environment has determined that the entry that is being looked up exists on a remote source or intermediate target other that the current computational environment. The reference to the remote object is provided to the requester.

For example, it would be beneficial to have a lookup for an object in the target environment, maintain the semblance of the target environment from which it came. An EJB on the target could use lookup mechanisms that are appropriate for the source environment, but not for the target environment. The BCE would provide the bridging back to the source environment or intermediate target environment, giving the local object the impression that it is running in the source environment.

In one embodiment, the lookup could be into a flat file that is edited when an asset is deployed to the local environment, and that asset needs to have a reference to a remote object. In a preferred embodiment, the lookup information would be persistent in a database table in the target environment.

In step 2820, the method determines if a JDBC fault has occurred. If a JDBC fault has occurred, the method continues to step 2825. Otherwise, the method continues to step 2830.

In step 2825, the method triggers the immediate distribution or synchronization of EB, ED, or RD assets.

For example, when an application program attempts to perform a database operation on a particular table, the actual changes to the table would be routed to the source or intermediate target environment.

In step 2830, the method determines if a server fault has occurred. A server fault occurs when a reference to an object on a server is attempted, and the server is registered as needing to have a proxy connection made to it. If so, the method proceeds to step 2835. Otherwise the method proceeds to step 2840.

In step 2835, the method looks up which CDS/ADS can proxy the request for the server being requested. The method then proceeds to utilize the CDS/ADS to proxy the communication to the server.

In one embodiment, the server would either be on the CDS/ADS or in the source environment.

For example, a CDS/ADS would have the capability to proxy IIOP requests to a server in the source environment behind a firewall. The proxy server on the CDS/ADS would handle the HTTP tunneling of the IIOP traffic to conform to the firewall specification for allowable traffic.

In step 2840, the method determines if an object fault has occurred. An object fault occurs when a request is made on an object that is a stub of the actual object. The stub acts as a proxy for the object, sending the request to the actual object. The actual object would typically reside in either a source environment or on an intermediate target. If an object fault has occurred, the method continues to step 2845. Otherwise the method continues to step 2850 where process 2800 ends.

In step 2845, the method executes the request using the stub object. The stub object, in turn, calls either into the source environment or to an intermediate target. In either of these environments exists the actual object that can fulfill the request. The actual object fulfills the request and the stub returns any result to the caller object.

Please note that the fulfillment of requests using a proxy object is well known in the prior art, especially middleware systems. The mechanism of a stub object fulfilling requests differs here in that the middleware environment did not intend to have these specific object interactions happening on the client or even an intermediate target. The capability to perform this request proxy behavior in a distributed fashion is only partially handled, if at all, by exiting middleware technology.

In an alternate embodiment, a check is made to determine if an asset adapter will handle this fault and a proxy request or redirection is performed to access the source asset. A proxy object accepts the request, calls the CDS/ADS, which in turn calls the EDA on the source, which performs the call within the source environment, and returns the result to the CDS/ADS, which returns the result to the CDA, and the CDA returns the result to the proxy object, the proxy object returns the results to the caller.

In a preferred embodiment, the package specification for different asset types also includes the capability to translate the directory entries for those asset types. In some cases, the directory entry would be a pointer to a reference to a resource in the EIS. The reference would need to be translated into a reference to a resource local to the target environment or to some intermediate target environment. A third possibility is that the reference would need to be translated to a server on the EIS, the resource may be accessed in a different way when outside the EIS (i.e. firewall traversal).

Note, the package can be incomplete, not defining all the assets that make up the call graph. In this case, the base environment on the target environment bridges the calls that attempt to access resources and code in the source environment.

Additional note, the package can be split into two or more packages. These packages can be deployed in different target environments. The base environment bridges the calls that attempt to access resources and code outside its node.

Quality of Service (QoS) in the system of Figure 9 (Distributed Internet Services – DIS) refers to the capacity to route the distribution of assets in a variable manner throughout the network. Certain assets are deemed higher priority or more secure than other assets. Those privileged assets are routed differently than other assets.

In the case of priority, the privileged asset would preempt the processing of normal assets, be routed ahead of lower priority assets, and/or execute with a higher priority in computational environments.

In the case of security, the privileged asset would be sent over a more secure channel, encrypted in the cache, and/or stored in the computational environment in a secured manner.

For example, two users are competing for the same Internet resources, yet there is no differentiation between these users or their uses of the Internet. The casual user is streaming MP3 file and chatting with friends in a chat room. The corporate user is attempting to conduct research while preparing a report for a meeting the next morning. There is limited bandwidth available, and each user is awarded bandwidth in the same manner, at the same priority. The casual user is using several Internet applications simultaneously with a fragmented interest in each application. The corporate user is focused on one application at a time, finding the needed information and moving onto the next task. If these applications utilized the QoS capabilities in the DIS, the casual user could have assets delivered at a lower priority than the corporate user. The assets for the corporate user would preempt the casual user's assets as each is being distributed. Use of intermediate computational environments would favor the processing for the corporate user over the casual user. The corporate user's traffic can be monitored and the corporate user can be charged according to a different service level agreement than the casual user.

QoS capabilities work best when resource contention is handled in a consistent manner. In a preferred embodiment, the DIS handles network interaction from the source to the target, a complete end-to-end solution. In this way, DIS provides a QoS capability on top of other protocols. And as the number of applications using those sub-protocols decreases, the DIS applications enjoy a more effective QoS functionality.

Figure 29 describes the capability of the DIS to assign, measure, enforce, and report Quality of Service (QoS) functionality throughout the DIS system. One of the objectives of QoS is to improve performance for higher priority applications by queuing and scheduling data as it is sent over a network. DIS keeps track of the QoS requirements and capabilities end-to-end: on the source tier, distribution tier, and target tier. The QoS requirements for assets are matched up to the QoS capabilities of the system to provide this functionality. The agents each participate in supporting the QoS capabilities by queuing traffic and scheduling the delivery of that traffic. As assets move through the system, they are either given space on a queue or space on a network connection (bandwidth).

Part of the QoS capabilities of DIS is routing the distribution of assets in a variable manner throughout the network. Certain assets are deemed higher priority or more secure than other assets. Those privileged assets are routed differently than other assets.

In the case of priority, the privileged asset would preempt the processing of normal assets, be routed ahead of lower priority assets, and execute with a higher priority in computational environments. This is accomplished by queuing the lower priority assets and delivering the higher-priority assets. The priority of an asset might be designated in the Other field 1179 of Figure 4.

In the case of security, the privileged asset would be sent over a more secure channel, encrypted in the cache, and/or stored in the computational environment in a secured manner.

In step 2910, an agent is about to process a set of assets or deliver those assets to another agent.

In step 2920, the agent examines the priority field, e.g. 1179, of the asset specification 1170 for the assets it is attempting to process or deliver to another agent.

In step 2930, the agent applies a quality of service algorithm to the prioritized list to determine which assets should be assigned resources on the delaying queue (step 2940), and which assets should be processed/delivered (step 2950).

In step 2940, lower priority assets are enqueued until the processing/delivery of higher priority assets is completed. As this method proceeds, the lower priority assets are either raised in priority by the QoS algorithm, or they are the highest priority for some round of evaluation.

In step 2950, higher priority assets are processed and/or delivered to other agents.

Note that Service Level Agreements (SLAs) are the contracts guaranteeing network capabilities. QoS enables the fulfillment of these SLA guarantees.

Also note, QoS is typically provided by differentiating data packets on a network. The data packets are correlated back to an application. The correlation allows the packets to be handled differently on the network. The correlation is done as an afterthought to provide QoS.

Note, DIS differs from the prior art by coupling the applications with an infrastructure that is aware of the different resources that constitute those applications. QoS granularity now can go beyond the data packets associated with an application. For example, QoS can be based on the DBMS data, moving the data at a higher priority than the other resources that make up an application.

Note, integration with metrics gathering software and hardware helps establish the resource loading and capabilities. Integration with the network devices (load balancers, switches, routers, network adapters, hubs) provides a connection between application and data packet handling.

Referring to Figure 31 which is a block diagram of a general system using the inventions described above, in various business situations. The system has one or more nodes 3130 connected to an optional intermediate target server (ITS1) 3160. This ITS1 could cache assets 240 used by a large number of client/target/nodes 3130, e.g. individual users in a large office building. Alternatively, a second ITS2 3170 could be located in a different physical location that happens to be close to a first pervasive device 3140 like a palm pilot, cell phone, or other device that has connection to the network 3110, e.g. by a radio, infrared, connection. A second pervasive device 3150 is connected to the network directly 3108, e.g. through any general connection like an internet service provider (ISP) and does not connect through an ITS (3160, 3170) One or more CDS/ADSs 3180 are connected to the network 3105 as well as one or more sources/EISs (910, 3190).

In one embodiment, the EIS has a publish 2200 agent, the CDS/ADS 3180 has a subscriber 2300 and caching 2500 agent, and the ITS 3160 has a computational 2400 and subscriber 2300 agent (and alternatively a cache agent 2500).

Referring back to Figure 30.

Steps 3002 to 3006 below describe examples of target and client nodes.

In step 3002, a deployment engineer or automated process defines the client workstation using a client node registry specification, sent to the subscriber agent 2300 on the CDS/ADS (960, 3180). The first client node is defined as TCN1 3130, and is a workstation within a small enterprise. TCN1 has the DIS subscriber 2300 and computational 2400 agents installed.

In step 3004, a deployment engineer or an automated process defines a wireless appliance client. This second client node is defined as TCN2 3140, and is a Palm Pilot that is able to

connect to the Internet over a wireless connection 3110. The Palm Pilot is capable of running the minimal application server on a Java virtual machine. The subscriber 2300 and computational 2400 agents are installed on the TCN2 3140 to provide the ability to distribute assets to the TCN2, and run those assets once they are deployed 1700 into the base environment 250 on TCN2 3140.

In step 3006, a deployment engineer or an automated process defines a wireless appliance client that is meant to have limited resources. This third client node is defined as TCN3 3150, a device with limited function, e.g., a Digital PCS Phone with limited web browser capabilities. TCN3 is heavily reliant on a point-of-presence in order to connect to a network, and can only be used to browse content. In this example, TCN3 does not have a Java virtual machine -- this means that it only acts as a web client. None of the Distributed Internet Services (DIS) 900 agents (2200, 2300, 2400, and 2500) are deployed to TCN3 3150.

Step 3007 continues to define other target nodes that may be on the network, as above.

Steps 3008 and 3010 below describe two example intermediate target servers.

In step 3008, a deployment engineer or an automated process defines an intermediate server 3160 to be used within the same small enterprise as TCN1 3130. The first intermediate server is defined as ITS1 3160, and is a server within a small enterprise that is intended to run part or all of a web application for one or more client nodes 3130 within the small enterprise. The subscriber 2300, computational 2400, and (optional) caching 2500 agents are installed on ITS1 3160.

In step 3010, a deployment engineer or an automated process defines an intermediate server to be used as both a point of presence for TCN3 3150 and as a CDS/ADS for TCN2 3140. The second intermediate server is defined as ITS2 3170. In a preferred embodiment, the ITS2 3170 can be a leased server resource provided by Application Service Provider (ASP) or ISP vendors with the DIS subscriber 2300, (optional) publisher 2200, computational 2400, and caching 2500 agents.

Other servers can be defined in a similar manner.

Note that when the deployment engineer or automated process defines a client and/or a server, this definition can be done by providing the defined computer with agents, e.g., subscription 2300, computation 2400, and/or cache 2500. These agents can be provided using the DIS and distributed these agents as packages of assets 240.

Steps 3012 to 3020 below describe definition of packages/applications as they are distributed across tiers of the network.

In step 3012, a package is defined for the end-user target environment such as TCN1 3130 and ITS1 3160, to enable the distribution of an optimal number of assets 240. For example, 20% of the application assets 240 that represents 80% of the time spent in the application by

the average user are distributed to the target. The deployment engineer creates the package specification 1100 explicitly, or employs the discover process 2100 to identify and characterize the execution of the application in the EIS (900, 3190).

In step 3014, a package 1100, PKG1 is defined for enterprise target environments such as ITS1 3160. PKG1 1100 defines an application (sub applications) that can support partial distribution of the application. As an example, some of the web pages will be generated by the computational agent 2400 on ITS1, some of the web pages will be generated back in the EIS.

In step 3016, a package is defined, PKG2 1100 is defined for end-user target environments such as TCN1 3130. PKG2 defines an application that can support partial distribution of the application (sub applications). As an example, some of the web pages will be generated by the computational agent 2400 on TCN1 3130, some of the web pages will be generated back in the EIS.

In step 3018, a package is defined, PKG3 1100 is defined for wireless access points such as ITS2 3170. The wireless access point, ITS2, is a server in a close proximity to the wireless device (3140, 3150). The intention with PKG3 is for assets to execute within the computational agent 2400 on ITS2 3170.

In step 3020, a package is defined, PKG4 1100 is defined for wireless devices such as TCN2 3140. The wireless device, TCN2, is connected 3110 to the Internet, through the wireless access point ITS2. The intention with PKG4 is for assets 240 to execute both within the computational agent 2400 on TCN2 3140 and within the computational agent 2400 on ITS2.

Similar steps are performed to define other end-user targets.

Steps 3022 to 3036 below describe various preferred ways assets 240 are distributed to computational environments and secondary caches across the tiers of the network.

In step 3022, the CDS/ADS initiates the distribution (1200, 1300) based on the schedule defined for package PKG1 1100. The subscription agent 2300 in the CDS/ADS requests version information from the publishing agent 2200 in the EIS. The version information is compared with the version information for assets 240 already delivered to ITS1, as described above. If there are assets 240 that have not been delivered, the subscription agent 2300 on the CDS/ADS requests those assets from the publishing agent 2200 on the EIS. The publishing agent 2200 exports the requested assets and returns them to the subscription agent 2300 on the CDS/ADS. The subscription agent 2300 on the CDS/ADS requests that the caching agent 2500 on the CDS/ADS store the assets. The caching agent 2500 on the CDS/ADS performs processing on all the new assets in the asset cache.

In step 3024, the subscription agent 2300 on ITS1 queries the subscription agent 2300 on CDS/ADS 3180 for new assets/PKG1 1100. The caching agent 2500 on ITS1 performs any targeted processing 1900 on the assets 240 that are to be delivered to TCN1. TCN1 requests

that the computational agent 2400 on ITS1 deploy the assets 240 into the base environment 250 on TCN1.

In step 3026, do the same as steps 3022 and 3024, except for PKG2 1100 into the asset cache on ITS1.

In step 3028, the subscription agent 2300 on TCN1 queries the subscription agent 2300 on ITS1 for new assets 240, e.g. PKG2 1100. New assets, associated with PKG2 1100, are distributed down 1400 by the subscription agent 2300 on ITS1 to the subscription agent 2300 on TCN1. The subscription agent 2300 on TCN1 requests that the computational agent 2400 on TCN1 deploy 1700 the assets into the base environment 250.

In step 3030, similar to step 3022 and 3024, except for PKG3 1100 into the asset cache on CDS/ADS.

In step 3032, similar to step 3024, except PKG3 1100 replaces PKG2 and ITS2 replaces ITS1.

In step 3034, similar to step 3022 and 3024, except for PKG4 is relayed from the CDS/ADS 3180 asset cache, into the asset cache on ITS2.

In step 3036, similar to step 3024, except for PKG4 from ITS2 cache into computational environment on TCN2.

Steps 3038 to 3056 below discuss various ways that assets are executed in a distributed manner across tier of the network.

In step 3038, TCN1 may access a web page that is generated on the EIS server. This is the same as normal web applications today, and is still available to use, when using DIS, when appropriate.

In step 3040, TCN1 may access a web page from the small enterprise server ITS1. TCN1 would only need a web browser to access the application in this method over a standard client/server communication connection 3107.

In step 3042, TCN1 may access a web page from the computational agent on TCN1. In this case, no part of the application would be running over the network connections. This is because the assets 240 in the web page were distributed over connection 3101 and processed as described above.

In step 3044, TCN1 would access a web page that was generated by assets that are running on the computational agents TCN1, ITS1, and the EIS. Note that these agents all work together to maintain the communication between these assets and a combination of assets in these environments, up to and including all the assets are used to generate a single web page, e.g., using the bridge process explained above.

In step 3046, changes may occur in the computational environment on TCN1 that need to be synchronized in the EIS. The computational agent 2400 on TCN1 identifies the synchronization asset and requests, through connection 3101 that the subscription agent 2300 on CDS/ADS synchronize 2000 the asset 240. The subscription agent 2300 on the CDS/ADS requests that the computational agent 2400 on the EIS synchronizes the synchronization asset.

In step 3048, TCN2 may access a web page from the small enterprise server ITS2. TCN2 would only need a web browser to access the application in this method. This is a standard known web page server connection 3110. However the assets used to create the small enterprise server on ITS2 were distributed from the EIS (900, 3190) over connections 3103 and 3104 using the agents as described above.

In step 3050, TCN2 may access a web page from the computational agent on TCN2. In this case, no part of the application would be running over the network connections. This is because the assets 240 in the web page were distributed over connection 3110 and processed as described above.

In step 3052, similar to 3044 but only bridging computation between TCN2 & ITS2

In step 3054, similar to 3046 but between TCN2 & ITS2

In step 3056, TCN3 3150 may access a web page from the small enterprise server ITS2. TCN3 would only need a web browser to access the application in this method using a well-known web page server connection 3108. However, the assets 240 in the web page were distributed over connection 3103 and 3104 and processed as described above.

Steps 3058 through 3062 below describe distribute of current versions of assets.

In step 3058, some outside process changes one or more assets on the EIS.

In step 3060, the CDS/ADS subscription agent 2300 requests that the EIS publishing agent 2200 checks the versions of assets on EIS.

In step 3062, the CDS/ADS determines if any assets have changed in PKG1...PKG4, if so the new asset(s) is distributed (1200, 1300, and 1400) to the appropriate target environment and asset cache.

## CLAIMS

We claim:

EE claims:

1. An extended environment data structure that is part of a digital asset, the digital asset capable of being transmitted over one or more multi-tiered networks, the data structure comprising:

one or more common descriptors that provide a unique identification of the digit asset on the networks;

one or more base environment descriptors that identify a required base execution environment on one or more target computers, the base execution environment being required to execute the digital asset on the respective target computer, where the base execution environment comprises zero or more other digital assets deployed to the respective target computer from one or more or the Enterprise Information Systems (EIS).

2. A data structure, as in claim 1, where the common descriptors include any one or more of the following: a digital asset name of the digital asset, a unique fully qualified name of the digital asset, an address of the digital asset, a size of the digital asset, a volatility descriptor of the digital asset, a common runnable descriptor of the digital asset, a user type descriptor of the digital asset, a security descriptor of the digital asset, a price descriptor of the digital asset, the independent deployment of the digital asset, and priority of the digital asset.

3. A data structure, as in claim 1, where the base environment include any one or more of the following: one or more DBMS, one or more application servers, one or more Web servers, one or more JMS implementations, one or more J2EE application servers, one or more browsers, one or more Java Virtual Machine (JVM) instantiations, one or more operating systems, systems, sub-systems, applications, sub-applications, components, modules, and functions.

4. A data structure, as in claim 1, where the whole is defined by any one or more of the following: a graph, a containment graph, a tube graph, a call graph, a pure representation expressible as a graph.

5. An extended environment data structure that is part of a digital asset, the digital asset capable of being transmitted over one or more multi-tiered networks, the data structure comprising:

one or more common descriptors that provide a unique identification of the digit asset on the networks;

one or more asset dependency descriptors that identify one or more associated digital assets, the associated digital assets being associated with the digital asset by means of a joint membership as parts of a whole; and

one or more base environment descriptors that identify a base execution environment on one or more target computers, the base execution environment being required to execute the digital asset on the respective target computer, where the base execution environment comprises zero or more other digital assets deployed to the respective target computer from one or more or the Enterprise Information Systems (EIS).

6. A data structure, as in claim 5, where the asset dependency descriptors include any one or more of the following: one or more names of other digital assets on which the digital asset is dependent, an asset identifier, and one or more unique fully qualified names of other digital assets on which the digital asset is dependent.

7. An extended environment data structure that is part of a digital asset, the digital asset capable of being transmitted over one or more multi-tiered networks, the data structure comprising:

one or more common descriptors that provide a unique identification of the digit asset on the networks;

one or more asset dependency descriptors that identify one or more associated digital assets, the associated digital assets being associated with the digital asset by means of a joint membership as parts of a whole;

one or more base environment descriptors that identify a base execution environment on one or more target computers, the base execution environment being required to execute the digital asset on the respective target computer; and

one or more EIS server dependencies descriptors that identify an EIS execution environment on the respective EIS on which the asset resides.

8. A data structure, as in claim 1, 5, and 7, where one or more of the EIS server dependencies descriptors identify an EIS execution environment on the respective EIS from which the asset is transformed into a neutral environment form.

9. A data structure, as in claim 1, 5, and 7, where one or more of the EIS server dependencies descriptors identify an EIS execution environment on the respective EIS from which the asset is prepared for transformation into a neutral environment.

10. A data structure, as in claim 1, 5, and 7, where the base execution environment comprises zero or more other digital assets deployed to the respective target computer from one or more or the Enterprise Information Systems (EIS).

11. A data structure, as in claim 1, 5, and 7, where one or more EIS server dependencies identified by one or more of the EIS server dependencies descriptors include one or more of the following: EIS operating systems, EIS database management systems (DBMS), EIS servers, EIS application servers, EIS web application servers, one or more general business applications, one or more accounting applications, customer relationship management systems (CRM), business to business (B2B) systems, supply chain management systems, business to customer (B2C) system, order fulfillment systems, electronic shopping systems, one or more Enterprise Application Integration systems, one or more legacy interfaces, one or more Java Connector Framework (JCF) connectors, one or more JCF connectors for legacy interfaces, and message oriented middleware applications.

12. A data structure, as in claim 11, where the EIS server dependencies include one or more of the following: one or more DBMS products, one or more Oracle DBMS, one or more Sybase DBMS, and one or more DB2 DBMS.

13. A data structure, as in claim 1, 5, and 7, where zero or more of the base environment descriptors and zero or more of the EIS server dependencies descriptors are capable of being compared independently to a neutral form in order to determine whether a transform of the digital asset is required for the asset to be deployed on the respective target computer.

14. A data structure, as in claim 1, 5, and 7, where one or more of the base environment descriptors and one or more of the EIS server dependencies descriptors are capable of being compared to determine whether a transform of the digital asset is required for the asset to be deployed on the respective target computer.

15. A data structure, as in claim 13 and 14, where the transform of the digital asset includes a transformation of data in a logic/data section of the digital asset.

16. A data structure, as in claim 1, 5, and 7, further comprising one or more reference descriptors.

17. A data structure, as in claim 16, where the reference descriptors include any one or more of the following: a reference link descriptor, a reference file descriptor, and a reference directory descriptor.

18. A data structure, as in claim 16, where the reference link descriptor provides a world-wide-web (WWW) address that has contents used for processing of the digital asset.

19. A data structure, as in claim 16, where the reference link descriptor provides a world-wide-web (WWW) address that has contents used during execution of the digital asset.

20. A data structure, as in claim 16, where the reference file descriptor is a unique fully qualified name of a file required for reference by the digital asset.

21. A data structure, as in claim 16, where the reference directory descriptor provides an additional address information that is used to locate one or more of the associated digital assets.

22. A data structure, as in claim 1, 5, and 7, further comprising one or more transform descriptors that enables a transform of the digital asset from the EIS execution environment to the base execution environment.

23. A data structure, as in claim 22, where the transform descriptor includes a properties descriptor that provides information required for use of the digital asset on an operating system of the base execution environment.

24.A data structure, as in claim 22, where the transform descriptor includes a format descriptor that provides information required for use of the digital asset on an operating system of the base execution environment.

25.A data structure, as in claim 22, where the transform descriptor includes a registry descriptor that provide information required for use of the digital asset on a Window's operating system on the base execution environment.

26.A data structure, as in claim 1, 5, and 7, further comprising one or more asset type descriptors.

27. A data structure, as in claim 26 where the asset type descriptors include any one or more of the following asset types: static content (SC), dynamic content (DC), enterprise Java beans (EJB), reference data (RD), session bean (SB), entity bean (EB), entity data (ED), Java class (JC), Java beans (JB), Java Connector Framework (JCF), and Java applet (JA).

28. A data structure, as in claim 1, 5, and 7, further comprising one or more asset category descriptors.

29. A data structure, as in claim 28, where the asset category descriptors include any one or more of the following: a presentational descriptor, a transactional descriptor, an integration connector descriptor, and a relational data descriptor.

30. A data structure, as in claim 28, where the asset category descriptors include any one or more of the following: a content descriptor, a presentational component descriptor, a

transactional component descriptor, an integration connector component descriptor, an object data descriptor, and a relational data descriptor.

31. A data structure, as in claim 1, 5, and 7, further comprising one or more asset class descriptors.

32. A data structure, as in claim 31, where the asset class descriptors include any one or more of the following: base, Java, non-Java, language, and non-language.

33. A data structure, as in claim 1, 5, and 7, further comprising one or more package relationship descriptors that represents a part-whole relationship between the digital asset and one or more packages containing the digital asset.

34. A data structure, as in claim 33, where the package relationship descriptors represents at least the following three relationships in the part-whole relationship: a mandatory part-whole relationship, a shared part-whole relationship, and a root part-whole relationship.

35. A data structure, as in claim 1, 5, and 7, further comprising one or more security descriptors.

36. A data structure, as in claim 35, where the security descriptors include any one or more of the following functions: encryption, authentication, authorization, and access control.

37. A data structure, as in claim 1, 5, and 7, further comprising one or more runnable descriptors.

38. A data structure, as in claim 37, further comprising one or more of the runnable descriptors that include a neutral format that would enable the deferment of the assignment to a target execution environment for the digital asset.

39. A data structure, as in claim 37, further comprising one or more of the runnable descriptors that include a target execution environment for the digital asset.

40. A data structure, as in claim 37, further comprising one or more of the runnable descriptors that include an EIS execution environment and a target execution environment for the digital asset.

41. A data structure, as in claim 1, 5, and 7, further comprising one or more non-runnable descriptors.

42. A data structure, as in claim 41, further comprising one or more non-runnable descriptors that include description of the base execution environment for the digital asset.

43. A data structure, as in claim 41, further comprising one or more of the non-runnable descriptors that include a neutral format that would enable the deferment of the assignment to a target execution environment for the digital asset.

44. A data structure, as in claim 41, further comprising one or more non-runnable descriptors that include description of the EIS execution environment and the base execution environment for the digital asset.

45. A data structure, as in claim 1, 5, and 7, further comprising one or more personalization descriptors that enable the digital asset to be customized upon delivery to one or more of the base execution environments.

46. A data structure, as in claim 45, where the personalization descriptors include one or more data keys, being derived from a directory service, that establishes a linkage among data elements in the EIS execution environment.

47. A data structure, as in claim 46, where the directory service is one or more of the following in combination or a federated hierarchy: an LDAP server, Single- Sign-On service, and/or JNDI service.

48. A data structure, as in claim 46, where the linkage of the data elements resolve to a DBMS query in which one or more EIS databases are partitioned specifically for the needs of one or more target environments.

49. A data structure, as in claim 46, where the linkage of the data elements resolve to one or more content related assets that are partitioned specifically for the needs of one or more target environments.

50. A data structure, as in claim 46, where the linkage of the data elements resolve to one or more application related assets that are partitioned specifically for the needs of one or more target environments.

51. A data structure, as in claim 50, where the application related assets are one or more of the following: presentational components and/or transactional components.

52. A data structure, as in claim 50, where the application related assets are one or more of the following: JSP, Java Servlet, and/or Java EJB.

53. A data structure, as in claim 45, where the personalization descriptors include one or more data keys that establish a linkage among data elements in the EIS execution environment.

54. A data structure, as in claim 45, where the personalization descriptors include one or more data keys that establish a linkage among logic elements in the EIS execution environment.

55. A data structure, as in claim 1, 5, and 7, further comprising one or more pricing descriptors.

56. A data structure, as in claim 55, where the pricing descriptors describe information about any one or more of the following: a price, a price scheme (subscription, pay to own, pay to use, one time payment), a payment detail, payment method, a check description, a credit card description, and a credit card number.

57. A data structure, as in claim 1, 5, and 7, further comprising one or more target information descriptors.

58. A data structure, as in claim 57, where the target information descriptors describe any one or more of the following: well known user, an anonymous user, one or more user groups, an entire user group, a target machine, an identifiable segment of target machines a collection of target machines, an internet protocol address mask, and a group of target computers in a node collection structure.

59. A data structure, as in claim 1, 5, and 7, further comprising one or more schema descriptors.

60. A data structure, as in claim 59, where the schema descriptors provide information that describe any one or more of the following:

database table names and definitions, database column names and definitions, database key identifiers and value ranges, database view names and definitions, and other well known database schema elements.

61. A data structure, as in claim 1, 5, and 7, further comprising one or more metadata descriptors.

62. A data structure, as in claim 61, where the metadata descriptors provide information that describe any or more of the following:

repository object definitions, scope object definitions, module object definitions, operation object definitions, exception object definitions, constant object definitions, properties object definitions, attribute object definitions, relationship object definitions, type object definitions, and other well known metadata object definitions.

63. A data structure, as in claim 1, 5, and 7, further comprising one or more distribution logic descriptors, each having one or more transactions rules and one or more concurrency rules.

64. A data structure, as in claim 63, where the transactions rules specify any of a number and a frequency of times that the digital asset can be distributed to one or more target computers.

65. A data structure, as in claim 64, where the concurrency rules specify whether or not there are any restrictions on distribution of the digital asset with respect to the distribution of one or more other digital asset.

66. A data structure, as in claims 1, 5, and 7, where the data structure is received from one or more network connections.

67. A data structure, as in claims 1, 5, and 7, where the data structure is sent over one or more network connections.

68. A data structure, as in claims 1, 5, and 7, where the data structure is stored on one or more memories.

69. A data structure, as in claims 1, 5, and 7, where information in the data structure is modified at one or more locations on one or more networks as the digital asset is distributed over the networks.

70. An extended environment data structure that is part of a digital asset, the digital asset capable of being transmitted over one or more multi-tiered networks, the data structure comprising:

one or more common descriptor means for providing a unique identification of the digit asset on the networks;

one or more asset dependency descriptor means for identifying one or more associated digital assets, the associated digital assets being associated with the digital asset by means of a joint membership as parts of a whole; and

one or more base environment descriptor means for identifying a base execution environment on one or more target computers, the base execution environment being required to execute the digital asset on the respective target computer, where the base execution environment comprises zero or more other digital assets deployed to the respective target computer from one or more or the Enterprise Information Systems (EIS).

71. A computer program product that contains an extended environment data structure that is part of a digital asset, the digital asset capable of being transmitted over one or more multi-tiered networks, the data structure comprising:

one or more common descriptors that provide a unique identification of the digit asset on the networks;

one or more base environment descriptors that identify a base execution environment on one or more target computers, the base execution environment being required to execute the digital asset on the respective target computer, where the base execution environment

comprises zero or more other digital assets deployed to the respective target computer from one or more or the Enterprise Information Systems (EIS).

72. A computer program product, as in claim 71, further comprising:

one or more asset dependency descriptors that identify one or more associated digital assets, the associated digital assets being associated with the digital asset by means of a joint membership as parts of a whole.

73. A memory structure having an extended environment data structure that is part of a digital asset stored on the memory structure, the digital asset capable of being transmitted over one or more multi-tiered networks, the data structure comprising:

one or more common descriptors that provide a unique identification of the digit asset on the networks;

one or more base environment descriptors that identify a base execution environment on one or more target computers, the base execution environment being required to execute the digital asset on the respective target computer, where the base execution environment comprises zero or more other digital assets deployed to the respective target computer from one or more or the Enterprise Information Systems (EIS).

74. A memory structure, as in claim 73, further comprising:

one or more asset dependency descriptors that identify one or more associated digital assets, the associated digital assets being associated with the digital asset by means of a joint membership as parts of a whole.

Discovery Claims

1. A discovery process that identifies member objects of one or more computer system parts in an Enterprise Information System (EIS) and establishes at least one topographical relationship among the member objects, the processes executed by one or more computers, each of the computers having one or more memories and one or more central processing units (CPU), the process comprising the steps of:

    a. traversing one or more computer file systems of the EIS to find one or more of the member objects, the member objects meeting one or more selection criteria;

    b. for each member object found, placing a digital asset identifier of the member object in an intermediate representation, the intermediate representation being a graph with nodes and edges, each of the digital asset identifiers corresponding to one of the nodes of the graph, the edges representing the topographical relationship;

    c. creating a digital asset from the member object by placing the member object in a logic/data section of the digital asset and attaching an extended environment data structure to the logic/data section;

    d. storing the digital asset in an asset inventory container object; and

    e. repeating steps a through d until the intermediate representation sufficiently describes the computer system part, where the asset inventory container object being a sufficiently complete inventory of the digital assets of the computer system part that meet the selection criteria.

2. A process, as in claim 1, further comprising the inserted step d1, of creating an entry in an asset definition data structure, the entry having descriptions of one or more digital asset attributes of the digital asset, the asset definition data structure being a complete list of the digital assets of the computer system part that meet the selection criteria.

3. A process, as in claim 1, where the computer system parts are any one or more of the following: a computer system, a computer sub-system, an application, a sub-application, a module, and a function.

4. A process, as in claim 1, further comprising the step of storing one or more descriptors in the extended environment after the extended environment is created in step c.

5. A process, as in claim 4, where the descriptors include one or more common descriptors that provide a unique identification of the digit asset on the networks.

6. A process, as in claim 5, where the common descriptor includes one or more of the following: a digital asset name of the digital asset, a unique fully qualified name of the digital asset, an address of the digital asset, a unique hash value for the digital asset, a checksum for the digital asset, and a size of the digital asset.

7. A process, as in claim 4, where the descriptors include one or more asset dependency descriptors.

8. A process, as in claim 7, where the asset dependency descriptors include any one or more of the following: one or more names of other digital assets on which the digital asset is dependent, an asset identifier, and one or more unique fully qualified names of other digital assets on which the digital asset is dependent.

9. A process, as in claim 4, where the descriptors include one or more reference descriptors.

10. A process, as in claim 9, where the reference descriptors include any one or more of the following: a reference link descriptor, a reference file descriptor, and a reference directory descriptor.

11. A process, as in claim 10, where the reference link descriptor provides a world-wide-web (WWW) address that has contents used for processing of the digital asset.

12. A process, as in claim 10, where the reference link descriptor provides a world-wide-web (WWW) address that has contents used during execution of the digital asset.

13. A process, as in claim 10, where the reference directory descriptor provides an additional address information that is used to locate one or more of the associated digital assets.

14. A process, as in claim 10, where the reference file descriptor is a unique fully qualified name of a file required for reference by the digital asset.

15. A process, as in claim 4, where the descriptors include one or more runnable descriptors that indicate that the digit asset is a runnable digital asset.

16. A process, as in claim 15, where one or more of the runnable descriptors include a description of an neutral execution environment, enabling the dynamic targeting of the digital asset for one or more target environments.

17. A process, as in claim 15, where one or more of the runnable descriptors include a description of an EIS execution environment.

18. A process, as in claim 4, where the descriptors include one or more non-runnable descriptors that indicates that the digit asset is a non-runnable digital asset.

19. A process, as in claim 18, where one or more of the non-runnable descriptors include a description of an EIS execution environment.

20. A process, as in claim 18, where one or more of the non-runnable descriptors include a description of an neutral execution environment, enabling the dynamic targeting of the digital asset for one or more target environments.

21. A process, as in claim 2, where the asset definition data structure includes any one or more of the following: an asset identification, an asset location, a URL, a name, an asset type, and a version.

22. A discovery system that identifies one or more member objects of one or more computer system parts of an Enterprise Information System (EIS) and establishes at least one topographical relationship among the member objects, the system comprising:

a. a traversor that traverses one or more computer file systems to find one or more of the member objects, the member objects meeting one or more selection criteria;

b. an intermediate representation builder that, for each member object found by the traversor, places a digital asset identifier in an intermediate representation, the intermediate representation being a graph with nodes and edges, the digital asset identifier corresponding to one of the nodes of the graph, the edges representing the topographical relationship;

c. a digital asset creator that creates a digital asset from the member object by placing the data object in a logic/data section of the digital asset and attaching an extended environment data structure to the logic/data section; and

d. an inventory function that stores the digital asset in an asset inventory container object.

21. A discovery system that identifies member objects of one or more computer system parts of an Enterprise Information System (EIS) and establishes at least one topographical relationship among the member objects, the system comprising:

a. means for traversing one or more computer file systems to find one or more of the member objects, the member objects meeting one or more selection criteria;

b. means for placing a digital asset identifier in an intermediate representation for each member object found, the intermediate representation being a graph with

nodes and edges, the digital asset identifier corresponding to one of the nodes of the graph, the edges representing the topographical relationship;

c. means for creating a digital asset from the member object by placing the member object in a logic/data section of the digital asset and attaching an extended environment data structure to the logic/data section; and

d. means for storing the digital asset in an asset inventory container object.

22. A computer program product that contains a computer program with the following process steps:

a. traversing one or more computer file systems to find one or more of the member objects, the member objects meeting one or more selection criteria;

b. for each member object found, placing a digital asset identifier of the respective member object in an intermediate representation, the intermediate representation being a graph with nodes and edges, the digital asset identifier corresponding to one of the nodes of the graph, the edges representing the topographical relationship;

c. creating a digital asset from the data object by placing the member object in a logic/data section of the digital asset and attaching an extended environment data structure to the logic/data section; and

d. storing the digital asset in an asset inventory container object.

23. A computer memory storing a computer program with the following steps:

a. traversing one or more computer file systems to find one or more of the member objects, the member objects meeting one or more selection criteria;

b. for each member object found, placing a digital asset identifier of the member object in an intermediate representation, the intermediate representation being a graph with nodes and edges, the digital asset identifier corresponding to one of the nodes of the graph, the edges representing the topographical relationship;

c. creating a digital asset from the member object by placing the member object in a logic/data section of the digital asset and attaching an extended environment data structure to the logic/data section; and

d. storing the digital asset in an asset inventory container object.

24. A discovery process executed by a computer with one or more memories and one or more central processing units (CPUs), the process comprising the steps of:

a. determining a starting point in a sub part of one or more application programs, the sub part of the application program having a path of execution;

b. determining one or more edges of the path of execution and one or more elements of the sub part of the application program, each of the elements connected by at least one of the edges;

c. placing the elements in an asset candidate list;

d. classifying one or more of the elements in the asset candidate list according to an asset type; and

e. determining one or more of the elements in the asset candidate list that is to be included in an asset package.

25. A discovery process, as in claim 24, where the path of execution is any one or more of the following: a call graph, an execution call graph, a dependency tree, a set of one or more hyperlinks, an expressed call graph, and an implied call graph.

26. A discovery process, as in claim 24, where the edges are identified as one or more of the following: a hyperlink, a method call, a program call, a sub routine call, a program name in an execution list, and a call to an external program.

27. A discovery process, as in claim 24, where the sub part is an entire application program.

28. A discovery process, as in claim 27, where the application program is in the form of any one or more of the following: Perl, PHP, Java, Microsoft C##, C++, ASP, Visual Basic, Delphi, Fortran, a web page, and a Java Server Page (JSP).

29. A discover process, as in claim 24, where the edges are determined by a text search of the part of the application program.

30. A discovery process, as in claim 24, where the edges are determined by a text search of the part of the reverse engineered application program.

31. A discovery process executed by a computer with one or more memories and one or more central processing units (CPUs), the process comprising the steps of:

a. determining a top level page in a set of web pages, the set of web pages related to one another by a path of execution;

b. determining one or more hyperlinks as edges of the path of execution and one or more elements of the sub part of the application program, each of the elements connected by at least one of the hyperlinks;

c. placing the elements in an asset candidate list;

d. classifying one or more of the elements in the asset candidate list according to an asset type; and

e. determining one or more of the elements in the asset candidate list that is to be included in an asset package.

32. A discovery process, as in claim 31, where the top-level page is a Java Server Page (JSP).

33. A discovery process, as in claim 31, where the top-level page comprises one or more of the following: HTML and Java source code.

34. A discovery process, as in claim 31, where the hyperlinks are identified by a text search of the top-level page.

35. A discovery process, as in claim 31, where the elements are java objects that the top-level page will instantiate.

36. A discovery process, as in claim 35, where the asset types are determined by which classes of java objects are loaded by a servlet and the classes are mapped by a Java Servlet specification.

37. A discovery process, as in claim 31, where one or more of the elements has a static HTML asset type.

38. A discovery process, as in claim 33, where the static HTML text of the static HTML asset type has one or more image tags and an additional element is created in the asset candidate list for one or more of the image tags.

39. A discovery process, as in claim 38, where the image tag includes any one or more of the following: IMG and A.

40. A discovery process, as in claim 31, where one or more of the elements has an asset type of a java class file (JC).

41. A discovery process, as in claim 40, that generates an external call graph for the JC, the external call graph for the JC being a list of method calls that are made within a class representation of the JC.

42. A discovery process, as in claim 41, where the external call graph for the JC is generated by any one or more of the following: byte code examination and de-compilation technology.

43. A discovery process, as in claim 42, where one or more of the method calls of the external call graph for the JC creates an element in the asset candidate list.

44. A discovery process, as in claim 31, where one or more of the elements has a JAR file asset type.

45. A discovery process, as in claim 44, where the JAR file has any one of the following file extensions: "JAR", "WAR", and "EAR".

46. A discovery process, as in claim 44, that generates an external call graph for each java class in the JAR file, the external call graphs for the java classes in the JAR file being a list of JAR method calls that are made within a class representation of the JAR file.

47. A discovery process, as in claim 44, where the external call graph for the JAR file is generated by any one or more of the following: byte code examination and de-compilation technology.

48. A discovery process, as in claim 31, where one or more of the elements has an EJB asset type.

49. A discovery process, as in claim 48, further comprising the steps of:

a. matching a method signature against an interface of one or more of the EJB's; and

b. adding the EJB digital asset to the asset candidate list if there is no match.

Export Claims:

1. An export process for creating one or more preliminary packages of one or more digital assets, the processing comprising the steps of:

traversing an intermediate representation of one or more parts of a computer system while applying one or more context rules to determine a context of the parts, the intermediate representation being a graph with nodes and edges, each of the nodes of the graph corresponding to an identifier for a digital asset, the edges representing a topographical relationship among the digital assets of the parts, the context being a standard specified context;

performing a directed search for any of the set of runnable member objects and non-runnable member objects in one or more locations in an Enterprise Information System (EIS) identified in the intermediate representation and specified by the context;

accessing one or more of the set of runnable member objects and of the non-runnable member objects at their respective locations in the EIS identified in the intermediate representation and specified by the context;

building a preliminary package specification with one or more of the set of the runnable member objects and the non-runnable member objects; and

updating each of the digital assets in an asset inventory, the updated digital assets being listed in the preliminary package specification, the updating done by adding one or more export descriptors to an extended environment of the respective updated digital assets.

2. An export process, as in claim 1, further comprising updating one or more entries in an asset definition data structure corresponding to each of the updated digital assets.

3. An export process, as in claim 1, where the direct search is performed as a requirement of the specified context.

4. An export process, as in claim 1, where the directed search is for only one or more runnable member objects.

5. An export process, as in claim 1, where the directed search is for only one or more non-runnable member objects.

6. An export process for creating one or more preliminary packages of one or more digital assets, the processing comprising the steps of:

traversing an intermediate representation of one or more parts of a computer system while applying one or more context rules to determine a context of the parts, the intermediate representation being a graph with nodes and edges, each of the nodes of the graph

corresponding to an identifier for a digital asset, the edges representing a topographical relationship among the digital assets of the parts, the context being one of a standard specified context and a non-specified context;

if a standard specified context is determined, performing a directed search for any of the set of runnable member objects and non-runnable member objects in one or more locations in an Enterprise Information System (EIS) identified in the intermediate representation and specified by the context;

if a non-specified context is determined, performing an implicit traversal search for any of the set of runnable member objects and non-runnable member objects in one or more locations in an Enterprise Information System (EIS) identified in the intermediate representation;

accessing one or more of the set of runnable member objects and of the non-runnable member objects at their respective locations in the EIS identified in the intermediate representation and specified by the context if standard specified context is determined;

building a preliminary package specification with one or more of the set of the runnable member objects and the non-runnable member objects; and

updating each of the digital assets in an asset inventory, the updated digital assets being listed in the preliminary package specification, the updating done by adding one or more export descriptors to extended environment of the respective updated digital assets.

7. An export process, as in claim 6, further comprising the step of updating one or more entries in an asset definition data structure corresponding to each of the updated digital assets.

8. An export process for creating one or more preliminary packages of one or more digital assets, the process comprising the steps of:

traversing an intermediate representation of one or more parts of a computer system while applying one or more context rules to determine if there is a non specified of the parts, the intermediate representation being a graph with nodes and edges, each of the nodes of the graph corresponding to an identifier for a digital asset, the edges representing a topographical relationship among the digital assets of the parts;

performing an implicit traversal search for any of the set of runnable member objects and non-runnable member objects in one or more locations in an Enterprise Information System (EIS) identified in the intermediate representation;

accessing one or more of the set of runnable member objects and of the non-runnable member objects at their respective locations in the EIS identified in the intermediate representation;

building a preliminary package specification with one or more of the set of the runnable member objects and the non-runnable member objects; and

updating each of the digital assets in an asset inventory, the updated digital assets being listed in the preliminary package specification, the updating done by adding one or more export descriptors to extended environment of the respective updated digital assets.

9. An export process, as in claims 1, 6, and 8, where the descriptors include one or more common descriptors that provide a unique identification of the digit asset on the networks.

10. A export process, as in claim 9, where the common descriptor includes one or more of the following: a common security export descriptor and a common runnable descriptor.

11. An export process, as in claims 1, 6, and 8, where the export descriptors include one or more dependency descriptors.

12. An export process, as in claims 1, 6, and 8, where the dependency descriptor includes one or more asset dependency descriptors.

13. An export process, as in claims 1, 6, and 8, where the dependency descriptor includes one or more EIS server dependency descriptors.

14. An export process, as in claim 13, where the EIS server dependencies include one or more of the following: EIS operating systems, EIS database management systems (DBMS), EIS servers, EIS application servers, EIS web application servers, one or more general business applications, one or more accounting applications, enterprise resource planning systems (ERP), customer relationship management systems (CRM), business to business (B2B) systems, supply chain management systems, business to customer (B2C) system, order fulfillment systems, electronic shopping systems, and message oriented middleware applications.

15. An export process, as in claim 13, where the EIS server dependencies include one or more of the following: one or more Oracle DBMS, one or more Sybase DBMS, one or more DB2 DBMS, one or more Microsoft SQLServer DBMS, one or more Postgres DBMS, one or more Informix DBMS, one or more Informix Cloudscape DBMS, one or more HypersonicSQL DBMS.

16. An export process, as in claims 1, 6, and 8, where the export descriptors include one or more type descriptors.

17. An export process, as in claims 16, where the type descriptors include any one or more of the following: static content (SC), dynamic content (DC), enterprise java beans (EJB), reference data (RD), session bean (SB), entity bean (EB), Java Connector Framework connectors (JCF), entity data (ED), java class (JC), and java beans (JB).

18. An export process, as in claims 1, 6, and 8, where the export descriptors include one or more asset category descriptors.

19. An export process, as in claim 18, where the asset category descriptors include any one or more of the following: a content descriptor, a presentational descriptor, a transactional descriptor, and a relational data descriptor.

20. An export process, as in claims 1, 6, and 8, further comprising one or more asset class descriptors.

21. An export process, as in claim 20, where the asset class descriptors include any one or more of the following: base, java, non-java, language, and non language.

22. An export process, as in claims 1, 6, and 8, further comprising one or more schema descriptors.

23. An export process, as in claim 22, where the schema descriptors provide information that describe any or more of the following: database table names and definitions, database column names and definitions, database key identifiers and value ranges, database view names and definitions, and other well known database schema elements.

24. An export process, as in claims 1, 6, and 8, further comprising one or more metadata descriptors.

25. An export process, as in claim 24, where the metadata descriptors provide information that describe any or more of the following: repository object definitions, scope object definitions, module object definitions, operation object definitions, exception object definitions, constant object definitions, properties object definitions, attribute object definitions, relationship object definitions, type object definitions, and other well known metadata object definitions.

26. An export process, as in claims 1, 6, and 8, further comprising one or more transform descriptors.

27. An export process, as in claim 26, where the transform descriptor describes a transformation of data in a logic/data section of the digital asset.

28. An export process, as in claim 26, where the transform descriptor describes a transformation of logic in a logic/data section of the digital asset.

29. An export process, as in claim 26, where the transform descriptors describe a transform of the digital asset from the EIS execution environment to the target base execution environment.

30. An export process, as in claim 26, where the transform descriptors describe a transform of the digital asset from the EIS execution environment to a neutral form that can be dynamically targeted for one or more target base execution environments.

31. An export process, as in claim 26, where the transform descriptor includes a properties descriptor that provides information required for use of the digital asset on an operating system of the base execution environment.

32. An export process, as in claim 26, where the transform descriptor includes a format descriptor that provides information required for use of the digital asset on an operating system of the base execution environment.

33. An export process, as in claim 26, where the transform descriptor includes a registry descriptor that provide information required for use of the digital asset on a Window's operating system on the base execution environment.

34. An export process, as in claim 26, where the transform descriptor includes on or more property files that provide information required for use of the digital asset on an operating system on the base execution environment.

35. An export process, as in claims 1, 6, and 8, further comprising one or more reference descriptors.

36. An export process, as in claim 35, where the reference descriptors include any one or more of the following: a reference link descriptor, a reference file descriptor, and a reference directory descriptor.

37. An export process, as in claim 36, where the reference link descriptor provides a world-wide-web (WWW) address that has contents used for processing of the digital asset.

38. An export process, as in claim 36, where the reference link descriptor provides a world-wide-web (WWW) address that has contents used during execution of the digital asset.

39. An export process, as in claim 36, where the reference file descriptor is a unique fully qualified name of a file required for reference by the digital asset.

40. An export process, as in claim 36, where the reference directory descriptor provides an additional address information that is used to locate one or more of the associated digital assets.

41. An export process, as in claims 1, 6, and 8, further comprising one or more security descriptors.

42. An export process, as in claims 41, where the security descriptor include any one or more of the following functions: encryption, authentication, authorization, and access control.

43. An export process, as in claims 1, 6, and 8, further comprising one or more package relationship descriptors that represent a part-whole relationship between the digital asset and one or more packages containing the digital asset.

44. An export process, as in claim 43, where the package relationship descriptors represents at least the following three relationships in the part-whole relationship: a mandatory part-whole relationship, a shared part-whole relationship, and a root part-whole relationship.

45. An export process, as in claim 1, 6, and 8, further comprising one or more distribution logic descriptors, each having one or more transactions rules and one or more concurrency rules.

46. An export process, as in claim 45, where the transactions rules specify any of a number and a frequency of times that the digital asset can be distributed to one or more target computers.

47. An export process, as in claim 45, where the concurrency rules specify whether or not there are any restrictions on distribution of the digital asset with respect to the distribution of one or more other digital asset.

CDS Claims

We claim:

1. A component distribution server (CDS) system connected to one or more networks through one or more respective network interfaces, the CDS system comprising:

a package specification process that receives one or more packages, the packages being subparts of one or more application programs from one or more enterprise information systems (EIS), the packages having one or more assets, each asset having an asset type and two or more asset layers, a first asset layer being a logic/data layer and a second asset layer being an extended environment layer, the logic/data layer having information that embodies the function of the asset and the extended environment layer being a subset of the EIS and having portions of the EIS necessary to support the respective logic/data layer;

a process adapter process that translates one or more of the assets layers so that the asset can perform the asset function on one or more target base environments of one or more target computers; and

a target process that changes the one or more of the layers of the asset in order to provide specific information for one or more of the specific target computers, whereby a transformed asset is an asset that is translated by the process adapter process and changed by the target process.

2. A CDS system, as in claim 1, where one or more of the transformed assets are stored in one or more memories of the CDS system.

3. A CDS system, as in claim 1, further comprising:

an asset packaging process that packages one or more of the transformed assets into a deployment package and distributions each of the packages over one or more of the networks to one or more of the target computers, the transformed assets in one or more of the deployment packages being selected by one or more requirements of the respective target computers.

4. A CDS system, as in claim 3, where the transformed assets are selected for the respective deployment package by one or more set operations.

5. A CDS system, as in claim 3, further where the CDS system does not distribute transformed assets to the target computers when the respective transformed asset is already located on one or more target computer memories accessible by the target computer.

6. A CDS system, as in claim 3, where one or more of the transformed assets that are distributed to the targets requires that an asset be requested from one or more of the EIS.

7.A CDS system, as in claim 3, where one or more of the transformed assets distributed to the targets are located in one or more memories of the CDS system.

8. A CDS system, as in claim 3, where the transformed assets are distributed to one or more of the target systems in a push mode.

9. A CDS system, as in claim 3, where one or more of the assets are distributed to one or more of the target systems upon request from the respective target system.

10. A CDS system, as in claim 1, where one or more of the target systems are any one or more of the following: one of the EIS, an intermediate server, an asset proxy cache server, a client, and an intermediate target.

11. A CDS system, as in claim 1, further comprising a scheduling process.

12. A CDS system, as in claim 11, where the scheduling process schedules the execution of one or more of the transformed packages on one or more of the target systems.

13. A CDS system, as in claim 11, where the scheduling process schedules the deletion of one or more of the transformed packages on one or more of the target systems.

14. A CDS system, as in claim 1, further comprising a versioning process that determines a current version of one or more of the assets on the respective EIS.

15. A CDS system, as in claim 14, where the versioning process causes the current version of the asset to be sent from the EIS to the CDS system, if the current version of the asset is not in one or more of the memories in the CDS system.

16. A CDS system, as in claim 1, further comprising a receiver section that receives received information from one or more of the target computers.

17. A CDS system, as in claim 16, where the received information includes changed information from one or more of the target systems and the CDS system relays the changed information to the respective EIS to synchronize the respective application program residing at the EIS.

18. A CDS system, as in claim 16, where the CDS system relays the received information to one or more other of the target systems.

19. A CDS system, as in claim 18, where the information is translated by the process adapter process and changed by the target process before being relayed.

20. A CDS system, as in claim 1, where one or more of the assets define a business algorithm that is part of an EIS algorithm in the application program.

21. A CDS system, as in claim 1, where one or more of the assets define a subset of a relational data base that is in one or more of the EIS.

22. A CDS system, as in claim 1, where one or more of the assets define an portion of an execution environment of the EIS, the portion of the execution environment required to execute one or more deployment packages on the respective target system.

23. A CDS system, as in claim 1, where one or more of the assets define a portion of an execution context of the EIS, the portion of the execution context required to execute one or more deployment packages on the respective target system.

24. A CDS system, as in claim 1, where one or more of the assets are session enterprise java bean assets (SB) and the transformation of the asset includes translating a deployment descriptor and generating a wrapper classes to enable the SB to be executed on the target system.

25. A CDS system, as in claim 1, where one or more of the assets are entity enterprise java bean assets (EB) and the transformation of the asset includes translating a deployment descriptor and generating a wrapper classes to enable the EB to be executed on the target system and specific partitioning and sub selection of EIS database data.

26. A CDS system, as in claim 25, further comprising a transformation that includes synchronizing changed information on the target system by transferring the changed information to a database management system resident on the respective EIS.

27. A CDS system, as in claim 1, where one or more of the assets are entity data assets (ED) and the transformation of the asset includes partitioning and sub selection of EIS database data.

28. A CDS system, as in claim 27, further comprising a transformation that includes synchronizing changed information on the target system by transferring the changed information to a database management system resident on the respective EIS.

29. A CDS system, as in claim 1, where one or more of the assets are reference data assets (RD) and the transformation of the asset includes partitioning and sub selection of EIS database data.

30. A CDS system, as in claim 29, where the sub selected data is read only data in the target system.

31. A CDS system, as in claim 1, where at least one of the assets is not a relational data asset type., presentation components, and static content.

32. A CDS system, as in claim 1, where at least one of the assets is not a presentation components asset type.

33. A CDS system, as in claim 1, where at least one of the assets is not static content asset type.

34. A process executed by a computer server connected to one or more networks, the process comprising the steps of:

receiving one or more packages from one or more enterprise information systems (EIS), the packages being subparts of one or more application programs, the packages having one or more assets, each asset having an asset type and two or more asset layers, a first asset layer being a logic/data layer and a second asset layer being an extended environment layer, the logic/data layer having information that embodies the function of the asset and the extended environment layer being a subset of the EIS and having portions of the EIS necessary to support the respective logic/data layer;

translating one or more of the assets layers so that the asset can perform the asset function on one or more target base environments of one or more target computers; and

changing one or more of the layers of the asset in order to provide specific information for one or more specific target computers.

35. A computer server comprising:

means for receiving one or more packages from one or more enterprise information systems (EIS), the packages being subparts of one or more application programs, the packages having one or more assets, each asset having an asset type and two or more asset layers, a first asset layer being a logic/data layer and a second asset layer being an extended environment layer, the logic/data layer having information that embodies the function of the asset and the extended environment layer being a subset of the EIS and having portions of the EIS necessary to support the respective logic/data layer;

means for translating one or more of the assets layers so that the asset can perform the asset function on one or more target base environments of one or more target computers; and

means for changing one or more of the layers of the asset in order to provide specific information for one or more specific target computers.

36. A computer memory storage device storing a computer program, the computer program comprising the steps of:

receiving one or more packages from one or more enterprise information systems (EIS), the packages being subparts of one or more application programs, the packages having one or more assets, each asset having an asset type and two or more asset layers, a first asset layer being a logic/data layer and a second asset layer being an extended environment layer, the logic/data layer having information that embodies the function of the asset and the extended

environment layer being a subset of the EIS and having portions of the EIS necessary to support the respective logic/data layer;

translating one or more of the assets layers so that the asset can perform the asset function on one or more target base environments of one or more target computers; and

changing one or more of the layers of the asset in order to provide specific information for one or more specific target computers.

Package Claims

We claim:

1. A package structure for distributing over one or more networks, the package structure comprising:

one or more first assets, each first asset having a first logic/data part and a first extended environment part, the first asset being having a type being one of following set: a relational data asset type and a static content asset type; and

one or more second assets, each second asset having a second logic/data part and a second extended environment part, the second asset being a presentation component asset type, the presentation component asset type used to generate content.

2. A package structure for distributing over one or more networks, the package structure comprising:

one or more first assets, each first asset having a first logic/data part and a first extended environment part, the first asset being having a type being a relational data asset type;

one or more second assets, each second asset having a second logic/data part and a second extended environment part, the second asset being a presentation component asset type, the presentation component asset type used to generate content; and

one or more third assets, each third asset having a third logic/data part and a third extended environment part, the third asset being a transaction component asset type.

3. A package structure for distributing over one or more networks, the package structure comprising:

one or more first assets, each first asset having a first logic/data part and a first extended environment part, the first asset being a relational data asset type;

one or more second assets, each second asset having a second logic/data part and a second extended environment part, the second asset being a presentation component asset type, the presentation component asset type used to generate content; and

one or more third assets, each third asset having a third logic/data part and a third extended environment part, the third asset being a static content asset type.

4. A package structure for distributing over one or more networks, the package structure comprising:

one or more first assets, each first asset having a first logic/data part and a first extended environment part, the first asset being a relational data asset type;

one or more second assets, each second asset having a second logic/data part and a second extended environment part, the second asset being a presentation component asset type, the presentation component asset type used to generate content;

one or more third assets, each third asset having a third logic/data part and a third extended environment part, the third asset being a static content asset type; and

one or more forth assets, each forth asset having a forth logic/data part and a forth extended environment part, the forth asset being a transaction component asset type.

5. A package structure for distributing over one or more networks, the package structure comprising:

one or more first assets, each first asset having a first logic/data part and a first extended environment part, the first asset being a static content asset type;

one or more second assets, each second asset having a second logic/data part and a second extended environment part, the second asset being a presentation component asset type, the presentation component asset type used to generate content; and

one or more third assets, each third asset having a third logic/data part and a third extended environment part, the third asset being a transaction component asset type.

6. A package structure for distributing over one or more networks, the package structure comprising:

one or more first assets, each first asset having a first logic/data part and a first extended environment part, the first asset being having a type being one of following set: a relational data asset type and a static content asset type; and

one or more second assets, each second asset having a second logic/data part and a second extended environment part, the second asset being a presentation component asset type, the presentation component asset type used to generate content and where the presentation component asset type includes any one or more of the following: a dynamic content asset, an entity bean, a session bean, and a java bean.

7. A package structure, as in claims 1, 2, 3, 4, or 5, that is a sub application of an application on one or more Enterprise Information Systems (EIS).

8. A package structure, as in claims 1, 2, 3, 4, or 5, that is a non-proprietary sub application of a proprietary application on one or more Enterprise Information Systems (EIS).

9. A package structure, as in claims 1, 2, 3, 4, or 5, that has a package boundary defined by the boundaries of two or more of the assets in the package structure including one or more first boundaries of the first assets and one or more second boundaries of the second assets.

10. A package structure, as in claims 1, 2, 3, 4, or 5, having a package boundary that includes any one or more of the following: an application programming interface (API), a component boundary, a Internet protocol boundary, a logical program division in the application, one or more edges of an application program call graph for the application, an open standard API, a proprietary standard API, and an extension framework.

11. A package structure, as in claims 1, 2, 3, 4, or 5, having a package boundary that includes an open standard API that includes any one or more of the following: one or more TCP/IP Standard Libraries, one or more Standard C/C++ Libraries, one or more CORBA Libraries, one or more Java servlets, one or more Java server pages, one or more enterprise Java beans (EJB), a JDBC (Java database connectivity), a JMS (Java Messaging Service), a HTML (Hypertext Markup Language), an HTTP (Hypertext Transfer Protocol), and a WML (Wireless Markup Language).

12. A package structure, as in claims 1, 2, 3, 4, or 5, having a package boundary that includes a proprietary standard API including any one or more of the following: a J2EE, one or more Microsoft Foundation Classes (MFC), one or more COM/DCOM (component object model/distributed component object model), one or more Oracle Call Interfaces (OCI), one or more Oracle Pro*C Libraries, and one or more Informix ESQL C Libraries.

13. A package structure, as in claims 1, 2, 3, 4, or 5, that includes a package boundary that includes extension frameworks including: one or more Unix Shell Scripts, one or more Windows batch files, a CICS Job Control Language, a Microsoft Visual Basis Script, and a Visual Basic.

14. A package structure, as in claims 1, 2, 3, 4, or 5, where the second logic/data part includes any one or more of the following: a sub system, a sub application, a module, a function, a variable, a sub application of an EIS program, and a data structure.

15. A package structure, as in claims 1, 2, 3, 4, or 5, where the second logic/data part is an object-oriented language that includes any one or more of the following: one or more components, one or more object classes, one or more object methods, and one or more data members.

16. A package structure, as in claims 1, 2, 3, 4, or 5, where the second logic/date part is an object-oriented language that includes any one or more of the following: one or more java archive files (JARs), one or more Java packages, and one or more Java classes.

17. A package structure, as in claims 1, 2, 3, 4, or 5, where the second logic/date part is an Java 2 Enterprise Edition programming language object-oriented language component that

includes any one or more of the following: one or more Java Server Pages (JSPs), one or more Java Servlets, and one or more Enterprise Java Beans (EJBs).

18. A package structure, as in claims 1, 2, or 3, where the first asset type is a relation data asset type and the first logic/data part, in addition to the relational data, includes any one or more of the following: any type of data structure, a set of relational database data, a set of object-oriented database data, one or more serialized objects, a set of hierarchical database data, a subset of EIS data, one or more data sets extracted from one or more XML structures, and a flat file.

19. A package structure, as in claims 1, 2, 3, 4, or 5, where the first and second extended environment parts are sub sets of any one or more of the following: a respective Enterprise Information System (EIS) application, a client environment, and one or more intermediate server environments.

20. A package structure, as in claims 1, 2, 3, 4, or 5, further comprising a first asset interface capable of enabling communication between any of the first, second, and third extended environment parts and one or more base environments on one or more target computers.

21. A package structure, as in claims 1, 2, 3, 4, or 5, where one or more of the assets is the package is a base environment.

22. A package structure, as in claims 1, 2, 3, 4, or 5, where one or more of the assets in the package is a base environment that includes any one or more of the following: a web server for static content assets (SC), a servlet engine for JSPs (JSP), a Java Runtime Environment for Java class assets (JB), an application server for the EJB assets (SB and EB), and a DBMS for the data assets (RD, ED, and EB).

23. A package structure, as in claims 1, 2, 3, 4, or 5, where one or more of the assets in the package include any one or more of the following agents: a publishing agent, a subscriber adapter, a caching agent, and a computational agent.

24. A package structure, as in claim 1, 2, 3, 4, or 5, where the assets in the package structure include any one or more of the following adapters: a discovery adapter, a versioning adapter, an export adapter, a process adapter, a target adapter, a deployment adapter, a synchronization adapter, a bridging adapter, an adjustment adapter, a streaming adapter, a quality of service (QoS) adapter, and an asset packaging process.

25. A package structure, as in claim 1, where one or more of the first assets in the package has a reference data asset type with a logic/data part that is data based on one or more queries and with an extended environment that is table schema.

26. A package structure, as in claim 1, where one or more of the assets in the package has an entity data asset type with a logic/data part that is data based on one or more queries and with an extended environment that is table schema.

27. A package structure, as in claim 1, where one or more of the assets in the package has an asset adapter based on a CDS adapter asset type, with a logic/data part that is one or more asset adapter class files that each support one of the asset types, and with an extended environment part that contains versioning information.

28. A package structure, as in claim 1, where one or more of the assets in the package has an asset adapter based on a CDS adapter asset type, with a logic/data part that is one or more asset adapter class files that each support one of the asset types, and with an extended environment part that contains licensing information.

29. A package structure, as in claims 1 or 2, where one or more of the assets in the package has a java server page (JSP) asset type, with a logic/data part that is a JSP and with an extended environment that is one or more JNDI entries.

30. A package structure, as in claims 1 or 2, where one or more of the assets in the package has a java class file asset type, with a logic/data part that is java class file and an extended environment that is one or more JNDI entries.

31. A package structure, as in claims 1 or 2, where one or more of the assets in the package has a session bean asset type with a logic/data part that is an enterprise java bean (EJB) implementation and with an extended environment that is an EJB stub and skeleton deployment descriptor and JNDI entries.

32. A package structure, as in claims 1 or 2, where one or more of the assets in the package has a java entity bean asset type with a logic/data part that is an EJB implementation based on a query and the extended environment that is an EJB stub and skeleton deployment descriptor and JNDI entries.

33. A package structure, as in claims 1 or 2, where one or more of the assets in the package has a Smalltalk asset type with a logic/data part that contains a Smalltalk image.

34. A package structure, as in claims 1 or 2, where one or more of the assets in the package has a C++ asset type with a logic/data part that is an executable file and with an extended environment that is one or more registry entries or environment variables.

35. A package structure, as in claims 1 or 2, where one or more of the assets in the package has a C++ asset type with a logic/data part that is a dynamic link library (DLL) and with an extended environment that is one or more registry entries or environment variables.

36. A package structure, as in claims 1 or 2, where one or more of the assets in the package has a Perl asset type with a logic/data part that is Perl script and with an extended environment that is one or more environment variables.

37. A package structure, as in claims 1 or 3, where the third logic/data part of the static content asset type is an HTML page.

38. A package structure, as in claims 1 or 3, where the third logic/data part of the static content asset type is an HTML page and with an extended environment part that includes a web server alias.

39. A package structure, as in claims 1 or 3, where the third logic/data part of the static content asset type is any one or more of the following: a JPEG file, a GIF file, Java Applet, a SVG (Scalable Vector Graphics), a PDF (Portable Document Format), a TIFF (Tag Image File Format), an EPS (Encapsulated Postscript) file, a PNG (Portable Network Graphics) file, an XML file, a WML (Wireless Meta Language) file, a BMP (Bit Map) file, an XHTML (extended HTML) file, a DHTML (Dynamic HTML) file, an MPEG (Motion Picture Expert Group) file, an AVI file, and any static content transferable via an HTTP protocol.

40. A package structure, as in claims 1 or 3, where the third logic/data part of the static content asset type has an extended environment that contains a web server alias.

41. A package structure, as in claims 1 or 3, where one or more of the assets in the package has a music asset type with a logic/data part that is an MP3 file and with an extended environment that is one or more sets of album and track information.

42. A package structure, as in claims 1 or 3, where one or more of the assets in the package has a video asset type with a logic/data part that is an MPEG file and with an extended environment that is one or more production notes.

43. A package structure, as in claims 1 or 3, where one of the assets in the package has a document asset type with a logic/data part that is a PDF file and with an extended environment that is one or more authentication certificates.

44. A package structure for distributing over one or more networks, the package structure comprising:

one or more first assets, each first asset having a first logic/data part and a first extended environment part, the first asset being a relational data asset type being selected from any one or more of the following types: a reference data (RD) and an entity data (ED); and

one or more second assets, each second asset having a second logic/data part and a second extended environment part, the second asset being a presentation component asset type, being selected from any one or more of the following types: a dynamic content (DC), a java bean (JB), a session bean (SB), and an entity bean (EB).

45. A package structure, as in claim 44, further comprising one or more third assets, each third asset having a third logic/data part and a third extended environment part, the third asset a

transaction component asset type being selected from any one or more of the following set of types: an entity bean (EB), a java bean (JB), a session bean (SB), and a dynamic content (DC).

46. A package structure, as in claim 45, where the set of types further comprises: a reference data (RD), an entity data (ED), and a naming directory.

47. A package structure, as in claim 44, further comprising one or more third assets, each third asset having a third logic/data part and a third extended environment part, the third asset being a static content asset type.

48. A package structure for distributing over one or more networks, the package structure comprising:

one or more first assets, each first asset having a first logic/data part and a first extended environment part, the first asset having a first asset type being a relational data asset type being either a reference (RD) type and an entity data (ED) data type; and

one or more second assets, each second asset having a second logic/data part and a second extended environment part, the second asset having a second asset type not being a relational data asset type, being either a session bean type (SB), an entity bean type (EB) or a Java server page (JSP) type.

49. A package structure stored on one or more memories, the package structure comprising:

one or more first assets, each first asset having a first logic/data part and a first extended environment part, the first asset being a relational data asset type; and

one or more second assets, each second asset having a second logic/data part and a second extended environment part, the second asset being a presentation component asset type.

50. A package structure, as in claim 49, further comprising one or more third assets, each third asset having a third logic/data part and a third extended environment part, the third asset being a transaction component asset type.

51. A package structure, as in claim 49, further comprising one or more third assets, each third asset having a third logic/data part and a third extended environment part, the third asset being a static content asset type.

52. An article of manufacture being a package structure stored on one or more memories, the package structure comprising:

one or more first assets, each first asset having a first logic/data part and a first extended environment part, the first asset being a relational data asset type; and

one or more second assets, each second asset having a second logic/data part and a second extended environment part, the second asset being a presentation component asset type.

53. An article, as in claim 52, further comprising one or more third assets, each third asset having a third logic/data part and a third extended environment part, the third asset being a transaction component asset type.

54. An article, as in claim 52, further comprising one or more third assets, each third asset having a third logic/data part and a third extended environment part, the third asset being a static content asset type.

55. A package structure means comprising:

one or more first asset means, each first asset means having a first logic/data part and a first extended environment part, the first asset means being a relational data asset type; and

one or more second asset means, each second asset means having a second logic/data part and a second extended environment part, the second asset means being a presentation component asset type.

56. A package structure means, as in claim 55, further comprising one or more third asset means, each third asset means having a third logic/data part and a third extended environment part, the third asset means being a transaction component asset type.

57. A package structure, as in claim 55, further comprising one or more third asset means, each third asset means having a third logic/data part and a third extended environment part, the third asset means being a static content asset type.

58. A package structure for distributing over one or more networks, the package structure comprising:

one or more first assets, each first asset having a first logic/data part and a first extended environment part, the first asset being a static content asset type; and

one or more second assets, each second asset having a second logic/data part and a second extended environment part, the second asset being a presentation component asset type.

59. A package structure, as in claim 58, further comprising one or more third assets, each third asset having a third logic/data part and a third extended environment part, the third asset being a transaction component asset type.

60. A package structure for distributing over one or more networks, the package structure comprising:

one or more first asset means for having a first logic/data part and a first extended environment part, the first asset means being having a type being one of following set: a relational data asset type and a static content asset type; and

one or more second asset means for having a second logic/data part and a second extended environment part, the second asset being a presentation component asset type, the presentation component asset type used to generate content.

61. A computer system having one or more memories with a package structure stored on one or more of the memories, the computer system capable of distributing the package structure over one or more networks, the package structure comprising:

one or more first assets, each first asset having a first logic/data part and a first extended environment part, the first asset being having a type being one of following set: a relational data asset type and a static content asset type; and

one or more second assets, each second asset having a second logic/data part and a second extended environment part, the second asset being a presentation component asset type, the presentation component asset type used to generate content.

62. A computer program product storing a package structure on one or more memories of the computer program product, the package structure comprising:

one or more first assets, each first asset having a first logic/data part and a first extended environment part, the first asset being having a type being one of following set: a relational data asset type and a static content asset type; and

one or more second assets, each second asset having a second logic/data part and a second extended environment part, the second asset being a presentation component asset type, the presentation component asset type used to generate content.

63. A computer memory having a package structure stored on the memory, the package structure comprising:

one or more first assets, each first asset having a first logic/data part and a first extended environment part, the first asset being having a type being one of following set: a relational data asset type and a static content asset type; and

one or more second assets, each second asset having a second logic/data part and a second extended environment part, the second asset being a presentation component asset type, the presentation component asset type used to generate content.

64. A package structure for distributing over one or more networks, the package structure comprising:

one or more first assets, each first asset having a first logic/data part and a first extended environment part, the first asset being having a type being one of following set: a relational data asset type and a static content asset type; and

one or more second assets, each second asset having a second logic/data part and a second extended environment part, the second asset being a presentation component asset type, the presentation component asset type used to generate content.

J2EE Transaction Deployment

1. A computer system for transactional deployment of one or more components, the system comprising:

one or more J2EE application servers stored on one or more memories of the system and capable of being executed by one or more central processing unit (CPUs);

one or more J2EE applications capable of being executed on the J2EE application servers;

one or more J2EE application containers within the J2EE application servers;

one or more J2EE application container components within the respective J2EE application containers;

one or more delivered J2EE components, delivered to the J2EE application server over one or more tiers of one or more networks;

one or more logical connections to one or more databases; and

a sphere of control encompassing the J2EE application server, the J2EE applications, the J2EE application containers, the J2EE application container components, the delivered J2EE components, and the logical connections, the sphere of control managing a transactional deployment of the delivered J2EE components and an update of the databases to have data consistent with the J2EE application.

2. A computer system, as in claim 1, where the J2EE application container component is one or more web containers.

3. A computer system, as in claim 2, where the web container contains one or more of the following: static content, a Java Servlet, a Java Server Page (JSP), an HTML page, and an XML page.

4. A computer system, as in claim 1, where the J2EE application container components further include one or more Enterprise Java Bean (EJB) containers.

5. A computer system, as in claim 4, where the EJB container contains one or more enterprise java beans.

6. A computer system, as in claim 2, where the J2EE application container components further include one or more Enterprise Java Bean (EJB) containers.

7. A computer system, as in claim 6, where the EJB container contains one or more enterprise java beans.

8. A computer system, as in claim 1, where the sphere of control comprises the steps of:

opening the database;

attempting to deploy the data to the database;

determining whether the deployment of the data is successful;

deploying one or more files containing the delivered J2EE components into the J2EE application server;

determining whether delivered J2EE components are successfully deployed into the J2EE application containers; and

retaining a version of the J2EE components for subsequent deployment failures; and

disposing of a prior version of the delivered J2EE components and the data only if both the deployment of the data and the deployment into the J2EE application container components is successful.

9. A method for creating a sphere of control that includes a J2EE application server, one or more data objects, and one or more delivered J2EE components, the method comprising the steps of:

opening a database;

attempting to deploy the data object to the database;

determining whether the deployment of the data is successful;

deploying one or more files containing the delivered J2EE components into the J2EE application server;

determining whether delivered J2EE components are successfully deployed into one or more J2EE application container components; and

retaining a version of the J2EE components for subsequent deployment failures; and

disposing of a prior version of the delivered J2EE components and the data only if both the deployment of the data and the deployment into the J2EE application container components is successful.

10. A method, as in claim 9, where the deployment of the data object is a success and the deployment of the delivered J2EE components is unsuccessful and successful deployment of the database is rolled back.

11. A method, as in claim 9, where the deployment of the data object is unsuccessful and the deployment of the delivered J2EE components is successful and prior version of the delivered J2EE is restored.

12. A method, as in claim 9, where any one or more of the data object deployment and the delivered J2EE components is unsuccessful and no action is taken.

13. A method, as in claim 9, where any one or more of the data object deployment and the delivered J2EE components are unsuccessful and a set of failure rules are applied.

14. A method, as in claim 13, where the failure rules include any one or more of the following: a retry of the data object deployment, a retry of the delivered J2EE component deployment, a retry of the failing deployment, accessing an alternative source for the data of the failed data deployment, and accessing an alternative source for the component of the fail component deployment.

15. A computer system for transactional deployment of one or more components, the system comprising:

means for opening a database;

means for attempting to deploy the data object to the database;

means for determining whether the deployment of the data is successful;

means for opening one or more files containing the delivered J2EE components;

means for determining whether delivered J2EE components are successfully deployed into one or more J2EE application containers;

means for disposing of a prior version of the delivered J2EE components and the data only if both the deployment of the data and the deployment into the J2EE application container components is successful; and

means of retaining a backup copy of the current deployment, to be available as a log to rollback subsequent deployment failures.

16. A computer program product comprising a program with the following steps:

opening a database;

attempting to deploy the data object to the database;

determining whether the deployment of the data is successful;

deploying one or more files containing the delivered J2EE components into one or more J2EE application server;

determining whether delivered J2EE components are successfully deployed into one or more J2EE application containers;

retaining a version of the J2EE components for subsequent deployment failures; and

disposing of a prior version of the delivered J2EE components and the data only if both the deployment of the data and the deployment into the J2EE application container components is successful.

17. A computer memory device storing a computer program, the program having the steps of:

opening a database;

attempting to deploy the data object to the database;

determining whether the deployment of the data is successful;

deploying one or more files containing the delivered J2EE components into one or more J2EE application servers;

determining whether delivered J2EE components are successfully deployed into one or more J2EE application container components;

retaining a version of the J2EE components for subsequent deployment failures; and

disposing of a prior version of the delivered J2EE components and the data only if both the deployment of the data and the deployment into the J2EE application container components is successful.

Deploy-Manifest

1. A system for distributing an asset to a multi-tiered network node, comprising:

at least one network;

at least one distribution server coupled to the network;

at least one port;

at least one processor coupled to the port and the network; and

at least one memory, coupled to the processor, storing instructions adapted to be executed by the processor to:

determine at least one pending asset on the distribution server;

receive a pending notice associated with the pending asset from the distribution server;

receive an asset descriptor manifest from the distribution server, the asset descriptor manifest having at least one asset identifier and at least one offset, the asset identifier and the offset being associated with the pending asset;

store the asset descriptor manifest in the memory;

receive a first asset fragment associated with the pending asset from the distribution server, the first fragment having a first offset indicating an end of the first fragment;

store the first asset fragment in a memory;

mark the offset with the first offset; and

receive a second asset fragment associated with the pending asset, the second asset fragment starting at the offset.

2. The system according to claim 1, wherein the pending asset has a schedule time, the schedule time being at or before a current time.

3. The system according to claim 1, wherein the pending asset is not included in the asset descriptor manifest if the pending asset is stored in the memory.

4. The system according to claim 1, wherein a target identifier associates the pending asset with the node.

5. The system according to claim 1, further including instructions adapted to be executed by the processor to:

query the distribution server for the pending notice.

6. The system according to claim 5, wherein the pending notice does not indicate a pending asset.

7. The system according to claim 6, further including instructions adapted to be executed by the processor to:

query the distribution server for a subsequent pending notice after a time delay.

8. The system according to claim 7, further including instructions adapted to be executed by the processor to:

determine the time delay based on a deployment time associated with a next asset deployment sequence.

9. The system according to claim 7, further including instructions adapted to be executed by the processor to:

determine the time delay by polling the distribution server.

10. The system according to claim 7, further including instructions adapted to be executed by the processor to:

determine the time delay based on an algorithm and at least one performance requirement of an application associated with the pending asset.

11. The system according to claim 1, wherein the distribution server insures that the pending asset is a current version residing on a source node connected to the network.

12. The system according to claim 1, wherein the asset descriptor manifest includes a deployment time schedule and the pending asset is deployed into a base environment of the node according to the deployment time schedule.

13. The system according to claim 1, wherein the asset descriptor manifest includes a plurality of pending assets, a subset of pending assets, and a subset of dependent assets, the subset of pending assets being deployed into a base environment associated with the node after the plurality of dependent assets are received.

14. The system according to claim 1, wherein the pending asset is deployed into a base environment associated with the node if the pending asset is an independent asset.

15. The system according to claim 1, further including instructions adapted to be executed by the processor to:

send the marked asset descriptor manifest to the distribution server.

16. The system according to claim 1, further including instructions adapted to be executed by the processor to:

send a request for pending assets not yet deployed to the distribution server.

17. The system according to claim 1, further including instructions adapted to be executed by the processor to:

change the asset identifier associated with the pending asset after the pending asset is received.

18. The system according to claim 1, wherein the second asset fragment includes a header having the asset identifier.

19. A method for distributing an asset to a multi-tiered network node, comprising:

determining at least one pending asset on a distribution server;

receiving a pending notice associated with the pending asset from the distribution server;

receiving an asset descriptor manifest from the distribution server, the asset descriptor manifest having at least one asset identifier and at least one offset, the asset identifier and the offset being associated with the pending asset;

storing the asset descriptor manifest in a memory;

receiving a first asset fragment associated with the pending asset from the distribution server, the first fragment having a first offset indicating an end of the first fragment;

storing the first asset fragment in a memory;

marking the offset with the first offset; and

receiving a second asset fragment associated with the pending asset, the second asset fragment starting at the offset.

20. A system for distributing an asset to a multi-tiered network node, comprising:

means for determining at least one pending asset on a distribution server;

means for receiving a pending notice associated with the pending asset from the distribution server;

means for receiving an asset descriptor manifest from the distribution server, the asset descriptor manifest having at least one asset identifier and at least one offset, the asset identifier and the offset being associated with the pending asset;

means for storing the asset descriptor manifest in a memory;

means for receiving a first asset fragment associated with the pending asset from the distribution server, the first fragment having a first offset indicating an end of the first fragment;

means for storing the first asset fragment in a memory;

means for marking the offset with the first offset; and

means for receiving a second asset fragment associated with the pending asset, the second asset fragment starting at the offset.

21. A computer-readable medium that stores instructions for distributing an asset to a multi-tiered network node, the instructions adapted to be executed by a processor to perform the steps of:

determining at least one pending asset on a distribution server;

receiving a pending notice associated with the pending asset from the distribution server;

receiving an asset descriptor manifest from the distribution server, the asset descriptor manifest having at least one asset identifier and at least one offset, the asset identifier and the offset being associated with the pending asset;

storing the asset descriptor manifest in a memory;

receiving a first asset fragment associated with the pending asset from the distribution server, the first fragment having a first offset indicating an end of the first fragment;

storing the first asset fragment in a memory;

marking the offset with the first offset; and

receiving a second asset fragment associated with the pending asset, the second asset fragment starting at the offset.

22. A method for distributing assets to multi-tiered network nodes, comprising:

sending an asset deployment query from a target node to a content distribution server, the query including a target identifier;

searching an asset deployment queue for the target identifier;

indicating, to the target node, whether the target identifier was found in the deployment queue;

sending a request for an asset descriptor manifest from the target node to the content distribution server;

identifying at least one asset that needs to be distributed to the target node;

creating an asset manifest entry associated with each asset that needs to be distributed to the target node;

assembling the asset descriptor manifest from at least one asset manifest entry;

sending the asset descriptor manifest from the content distribution server to the target node;

determining at least one asset associated with at least one asset manifest entry;

sending a request for the asset from the target node to the content distribution server;

creating a data structure associated with the asset; and

sending the data structure from the content distribution server to the target node.

23. A method for distributing an asset to multi-tiered network node, comprising:

querying an asset deployment queue for an instance of a target identifier;

requesting an asset descriptor manifest;

receiving the asset descriptor manifest, the asset descriptor manifest including at least one asset manifest entry associated with the asset;

determining the asset associated with the asset manifest entry;

requesting the asset;

receiving a data structure associated with the asset.

24. The method of claim 23, wherein the asset manifest entry includes an asset identifier and an offset.

25. The method of claim 24, wherein the asset manifest entry includes an asset type, a cache name, and a timestamp.

26. The method of claim 23, wherein the data structure includes a complete instance of the asset.

27. The method of claim 23, wherein the data structure includes a first fragment associated with the asset.

28. The method of claim 27, further comprising:

requesting a second fragment associated with the asset;

receiving a data structure including the second fragment; and

requesting a next fragment associated with the asset until a complete instance of the asset is received.

29. The method of claim 23, wherein the asset descriptor manifest includes a plurality of asset manifest entries.

30. The method of claim 29, wherein the data structure includes a plurality of assets corresponding to a plurality of asset manifest entries.

31. The method of claim 26, further comprising:

storing the asset;

selecting a deploy asset adapter associated with the asset; and

deploying the asset.

32. The method of claim 30, further comprising:

storing the plurality of assets;

selecting a deploy asset adapter associated with each of the plurality of assets; and

deploying each of the plurality of assets.

33. A method for distributing an asset to multi-tiered network node, comprising:

servicing an asset deployment query from a target node;

receiving a request for an asset descriptor manifest from the target node;

identifying at least one asset that needs to be distributed to the target node;

creating an asset manifest entry associated with each asset that needs to be distributed to the target node;

assembling the asset descriptor manifest from at least one asset manifest entry;

sending the asset descriptor manifest to the target node;

receiving a request for at least one asset from the target node;

creating a data structure associated with the asset; and

sending the data structure to the target node.

34. The method of claim 33, wherein the asset manifest entry includes an asset identifier and an offset.

35. The method of claim 34, wherein the asset manifest entry includes an asset type, a cache name, and a timestamp.

36. The method of claim 33, wherein the data structure includes a complete instance of the asset.

37. The method of claim 33, wherein the data structure includes a first fragment of the asset.

38. The method of claim 37, further comprising:

receiving a request for a second fragment of the asset;

sending a data structure including the second fragment of the asset; and

receiving a request for a next fragment until a complete instance of the asset is sent.

39. The method of claim 33, wherein the asset descriptor manifest includes a plurality of asset manifest entries.

40. The method of claim 39, wherein the data structure includes a plurality of assets corresponding to a plurality of asset manifest entries.

41. The method of claim 33, wherein said servicing includes:

receiving a target identifier associated with the target node;

searching a deployment queue for the target identifier; and

indicating, to the target node, whether the target identifier was found in the deployment queue.

42. The method of claim 33, wherein said identifying includes:

receiving a target identifier associated with the target node;

creating a plurality of relational database views;

performing selections on a set of network database tables using the plurality of relational database views and the target identifier;

extracting at least one asset identifier associated with at least one asset that needs to be distributed to the target node.

43. The method of claim 35, wherein said creating a data structure includes:

identifying an asset manifest entry associated with the asset;

extracting a cache name and an offset from the asset manifest entry;

retrieving the asset from an archive associated with the cache name;

determining whether the asset is larger than a predetermined threshold;

dividing the digital asset into a plurality of fragments, each fragment being smaller than the predetermined threshold;

inserting a first fragment into the data structure; and

updating the offset to correspond to a boundary of the first fragment.

Deploy Adaptor

1. A method for deploying an asset into a base environment of a target node, the target node being connected to a network through a network connection, comprising the steps of:

determining if an asset type of the asset is at least one of a session bean type and an entity bean type; and

inserting at least one enterprise Java bean implementation class associated with the asset into a target archive in the base environment of the target node.

2. The method according to claim 1, wherein the target archive further includes at least one other enterprise Java bean implementation class.

3. The method according to claim 1, further including storing at least one stub and skeleton associated with the enterprise Java bean implementation class in the target archive.

4. The method according to claim 1, further including storing at least one client deployment descriptor located in an extended environment of the asset in the target archive.

5. The method according to claim 1, further comprising the step of:
determining whether a target table exists in at least one memory of the target node, the target table being associated with the asset type.

6. The method according to claim 5, wherein the target table does exist and further comprising the step of:
storing an asset table data, in the target table.

7. The method according to claim 5, wherein the asset is an enterprise bean having at least one asset table specified in a logic/data portion of the asset, the asset table having asset table data.

8. The method according to claim 7, wherein the target table does not exist and further comprising the steps of:
creating the target table; and
storing an asset table data, located in the asset, in the target table.

9. The method according to claim 8, wherein said creating the target table includes:
using a target table specification, located in an extended environment portion of the asset, to create the target table.

10. The method according to claim 1, further comprising the steps of:
determining if the asset type of the asset is not a session bean type or an entity bean type;
determining if the asset type of the asset is at least one of a reference data type and an entity data type; and
determining whether a target table exists in at least one memory of the target node, the target table being associated with the asset type.

11. The method according to claim 10, wherein at least one data structure of the asset is stored in the target table.

12. The method according to claim 10, wherein the target table does not exist and further comprising the steps of:
creating the target table; and
storing at least one data structure of the asset in the target table.

13. The method according to claim 10, further comprising the steps of:
writing an entry in a target database table, the entry indicating that at least one data structure in the asset was stored on at least one memory of the target node.

14. The method according to claim 1, further comprising the step of:
determining if said deploying should occur based on a scheduled deployment time, the scheduled deployment time being at, or before, a current time.

15. A method for deploying an asset into a base environment of a target node, the target node being connected to a network through a network connection, comprising the steps of:

determining if an asset type of the asset is at least one of a static content type, a Java server page type, and a Java bean type, the asset being received through the network connection; and

copying at least one file specified in a data structure of the asset into a memory location on the target node corresponding to the asset type.

16. The method according to claim 15, further comprising the steps of:
determining if the asset type of the asset is not a static content type, a Java server page type, or a Java bean type; and
determining if the asset type of the asset is at least one of a plurality of secondary asset types.

17. The method according to claim 16, wherein the memory location is associated with the asset type of the asset.

18. A target node system, the target node system being connected to a network through a network connection, comprising:

means for determining if an asset type of an asset is at least one of a session bean type and an entity bean type; and

means for inserting at least one enterprise Java bean implementation class associated with the asset into a target archive in the base environment of a target node.

19. The target node system according to claim 18, further comprising:
means for storing, on at least one memory of the target node system, at least one of an EJB implementation class, stubs and skeletons associated with an EJB implementation class, and a target deployment descriptor located on an extended environment of the asset.

20. A medium for storing instructions adapted to be executed by a processor to perform the steps of:

determining if an asset type of an asset is at least one of a session bean type and an entity bean type; and

inserting at least one enterprise Java bean implementation class associated with the asset into a target archive in the base environment of a target node.

21. A method for deploying an asset to a multi-tiered network node, comprising:

determining an asset type associated with the asset;

selecting a target deployment asset adapter associated with the asset;

inserting at least one implementation class associated with the asset into an environment associated with the node; and

inserting an altered target deployment descriptor into the environment.

22. The method according to claim 21, wherein the asset is at least one of a session bean asset and an entity bean asset.

23. The method according to claim 21, wherein the asset is at least one of an entity bean asset, a reference data asset, and an entity data asset.

24. The method according to claim 21, wherein the implementation class is an enterprise Java bean implementation class.

25. The method according to claim 24, further comprising:
inserting at least one of an enterprise Java bean stub and an enterprise Java bean skeleton into the environment.

26. The method according to claim 25, wherein the at least one implementation class, enterprise Java bean stub, and enterprise Java bean skeleton is inserted into an archive in the environment.

27. The method according to claim 23, wherein the asset includes an associated data structure having a table specification.

28. The method according to claim 27, further comprising:

determining whether a database table associated with the table specification exists in the environment; and

creating the database table if the database table does not exist.

29. The method according to claim 28, further comprising importing data from the data structure into the database table.

30. The method according to claim 29, further comprising exporting the data from the data structure into a system database table.

31. A computer-readable medium that stores instructions for deploying an asset to a multi-tiered network node, the instructions adapted to be executed by a processor to perform the steps of:

determining an asset type associated with the asset;

selecting a target deployment asset adapter associated with the asset;

inserting at least one implementation class associated with the asset into an environment associated with the node; and

inserting an altered target deployment descriptor into the environment.

32. The computer-readable medium according to claim 31, wherein the asset is at least one of a session bean asset and an entity bean asset.

33. The computer-readable medium according to claim 31, wherein the asset is at least one of an entity bean asset, a reference data asset, and an entity data asset.

34. The computer-readable medium according to claim 32, wherein the implementation class is an enterprise Java bean implementation class.

35. The computer-readable medium according to claim 34, further comprising inserting at least one of an enterprise Java bean stub and an enterprise Java bean skeleton into the environment.

36. The computer-readable medium according to claim 35, wherein the at least implementation class, enterprise Java bean stub, and enterprise Java bean skeleton is inserted into an archive in the environment.

37. The computer-readable medium according to claim 33, wherein the asset includes an associated data structure having a table specification.

38. The computer-readable medium according to claim 37, further comprising instructions adapted to be executed by the processor to perform the steps of:

determining whether a database table associated with the table specification exists in the environment; and

creating the database table if the table does not exist.

39. The computer-readable medium according to claim 38, further comprising instructions adapted to be executed by the processor to perform the step of:

importing data from the data structure into the database table.

40. The computer-readable medium according to claim 39, further comprising instructions adapted to be executed by the processor to perform the step of:

exporting the data from the data structure into a system database table.

41. A system for synchronizing an asset on a multi-tiered network, comprising:

a processor;

a port coupled to the processor; and

a memory, coupled to the processor, storing instructions adapted to be executed by the processor to:

determine an asset type associated with the asset,

select a client deployment adapter associated with the asset,

insert at least one implementation class associated with the asset into an environment associated with the node, and

insert an altered target deployment descriptor into the environment.

42. The system according to claim 41, wherein the asset is at least one of a session bean asset and an entity bean asset.

43. The system according to claim 41, wherein the asset is at least one of an entity bean asset, a reference data asset, and an entity data asset.

44. The system according to claim 42, wherein the implementation class is an enterprise Java bean implementation class.

45. The system according to claim 44, further comprising inserting at least one of an enterprise Java bean stub and an enterprise Java bean skeleton into the environment.

46. The system according to claim 45, wherein the at least implementation class, enterprise Java bean stub, and enterprise Java bean skeleton is inserted into an archive in the environment.

47. The system according to claim 43, wherein the asset includes an associated data structure having a table specification.

48. The system according to claim 47, further storing instructions adapted to be executed by the processor to perform the steps of:

determining whether a database table associated with the table specification exists in the environment; and

creating the database table if the table does not exist.

49. The system according to claim 48, further storing instructions adapted to be executed by the processor to perform the step of:

importing data from the data structure into the database table.

50. The system according to claim 49, further storing instructions adapted to be executed by the processor to perform to perform the step of:

exporting the data from the data structure into a system database table.

Translation/Process

1. A system for translating an asset into a target environment format, comprising:

a processor; and

a memory, coupled to the processor, storing at least one asset, the asset having a logic/data section and an extended environment section, the logic/data section defining a function of the asset and an asset type and the extended environment section supporting the function of the asset in at least one source environment, the memory further storing instructions adapted to be executed by the processor to:

determine an asset type of the asset,

select a process asset adapter associated with the asset type and a target environment, and

translate the asset into a processed asset, the processed asset having a processed extended environment section supporting the function of the logic/data section in the target environment.

2. The system according to claim 1, further including instructions adapted to be executed by the processor to store the processed asset in the memory.

3. The system according to claim 1, wherein the target environment is a base environment deployed on a target system as at least one asset.

4. The system according to claim 1, wherein the source environment includes at least one of an Enterprise Information System, Windows 95, Windows NT, Weblogic, Unix, Linux, Websphere, iPlanet, and JBOSS.

5. The system according to claim 1, wherein the target environment includes at least one of Windows 95, Windows NT, Unix, Linux, JBOSS, Weblogic, Websphere, and iPlanet.

6. The system according to claim 1, wherein said selecting is based on at least one of the logic/data section, and the extended environment section.

7. The system according to claim 1, wherein said selecting is based on a descriptor associated with the target environment and a descriptor associated with the extended environment section.

8. The system according to claim 1, wherein the asset type is at least one of a session bean and an entity bean.

9. The system according to claim 8, further including instructions adapted to be executed by the processor to remove an EJB implementation from the logic/data section.

10. The system according to claim 9, further including instructions adapted to be executed by the processor to:

11.

generate a plurality of EJB stubs and skeletons associated with the target environment;

generate at least one class required by the target environment to execute the asset, the class having at least one associated translated deployment descriptor; and

move the EJB stubs and skeletons and the translated deployment descriptor into the asset.

11. The system according to claim 10, wherein the EJB stubs and skeletons and the translated deployment descriptors are moved into the extended environment section of the asset.

12. The system according to claim 11, further including a cache memory.

13. The system according to claim 12, further including instructions adapted to be executed by the processor to store the processed asset in the cache memory.

14. The system according to claim 12, further including instructions adapted to be executed by the processor to create at least one EJB proxy that is moved into the asset.

15. A method for translating an asset into a target environment format, comprising:

determining an asset type of the asset, the asset having a logic/data section and an extended environment section, the logic/data section defining a function of the asset and an asset type and the extended environment section supporting the function of the asset in at least one source environment;

selecting a process asset adapter associated with the asset type and a target environment, and

translating the asset into a processed asset, the processed asset having a processed extended environment section supporting the function of the logic/data section in the target environment.

16. The method according to claim 15, further including storing the processed asset in the memory.

17. The method according to claim 15, wherein the target environment is a base environment deployed on a target system as an asset.

18. The method according to claim 15, wherein the source environment includes at least one of an Enterprise Information System, Windows 95, Windows NT, Weblogic, Unix, Linux, Websphere, iPlanet, and JBOSS.

19. The method according to claim 15, wherein the target environment includes at least one of Windows 95, Windows NT, Unix, Linux, JBOSS, Weblogic, Websphere, and iPlanet.

20. The method according to claim 15, wherein said selecting is based on at least one of the logic/data section, and the extended environment section.

21. The method according to claim 15, wherein said selecting is based on a descriptor associated with the target environment and a descriptor associated with the extended environment section.

22. The method according to claim 15, wherein the asset type is at least one of a session bean and an entity bean.

23. The method according to claim 22, further comprising removing an EJB implementation from the logic/data section.

24. The method according to claim 23, further comprising:

generating a plurality of EJB stubs and skeletons associated with the target environment;

generating at least one class required by the target environment to execute the asset, the class having at least one associated translated deployment descriptor; and

moving the EJB stubs and skeletons and the translated deployment descriptor into the asset.

25. The method according to claim 24, wherein the EJB stubs and skeletons and the translated deployment descriptors are moved into the extended environment section of the asset.

26. The method according to claim 25, further including storing the processed asset in a cache memory.

27. The method according to claim 25, further including creating at least one EJB proxy that is moved into the asset.

28. A method for translating an asset for delivery to a multi-tiered network node, comprising:

determining an asset type associated with the asset;

selecting a process asset adapter associated with the asset;

retrieving a deployment descriptor from a data structure associated with the asset;

translating the deployment descriptor into a format associated with the node;

creating a plurality of class stubs and skeletons associated with the translated deployment descriptor; and

inserting the translated deployment descriptor and the plurality of class stubs and skeletons into the data structure associated with the asset.

29. The method of claim 28, wherein the asset is an entity bean type.

30. The method of claim 28, wherein the asset is a session bean type.

31. The method of claim 28, wherein the data structure includes a logic/data part and an extended environment part.

32. The method of claim 31, wherein the deployment descriptor is retrieved from the extended environment part.

33. The method of claim 31, wherein the translated deployment descriptor and the plurality of class stubs and skeletons are inserted into the extended environment part.

34. The method of claim 31, wherein said selecting the process asset adapter is based on the logic/data part and the format associated with the node.

35. The method of claim 32, wherein said selecting the process asset adapter is based on the deployment descriptor retrieved from the extended environment part.

36. The method of claim 31, further including removing an enterprise Java bean implementation from the logic/data part.

37. The method of claim 36, further including generating a plurality of classes associated with the plurality of class stubs and skeletons.

38. A computer-readable medium that stores instructions for translating an asset for delivery to a multi-tiered network node, the instructions adapted to be executed by a processor to perform the steps of:

determining an asset type associated with the asset;

selecting a process asset adapter associated with the asset;

retrieving a deployment descriptor from a data structure associated with the asset;

translating the deployment descriptor into a format associated with the node;

creating a plurality of class stubs and skeletons associated with the translated deployment descriptor; and

inserting the translated deployment descriptor and the plurality of class stubs and skeletons into the data structure associated with the asset.

39. A system for translating an asset for delivery to a multi-tiered network node, comprising:

a processor;

a memory, coupled to the processor, storing instructions adapted to be executed by said processor to:

determine an asset type associated with the asset;

select a process asset adapter associated with the asset;

retrieve a deployment descriptor from a data structure associated with the asset;

translate the deployment descriptor into a format associated with the node;

create a plurality of stubs and skeletons associated with the translated deployment descriptor; and

insert the translated deployment descriptor and the plurality of stubs and skeletons into the data structure associated with the asset.

40. A system for translating an asset into a target environment format, comprising:

means for determining an asset type of the asset, the asset having a logic/data section and an extended environment section, the logic/data section defining a function of the asset and an asset type and the extended environment section supporting the function of the asset in at least one source environment;

means for selecting a process asset adapter associated with the asset type and a target environment, and

means for translating the asset into a processed asset, the processed asset having a processed extended environment section supporting the function of the logic/data section in the target environment.

41. A system for translating an asset for delivery to a multi-tiered network node, comprising:

means for determining an asset type associated with the asset;

means for selecting a process asset adapter associated with the asset;

means for retrieving a deployment descriptor from a data structure associated with the asset;

means for translating the deployment descriptor into a format associated with the node;

means for creating a plurality of class stubs and skeletons associated with the translated deployment descriptor; and

means for inserting the translated deployment descriptor and the plurality of class stubs and skeletons into the data structure associated with the asset.